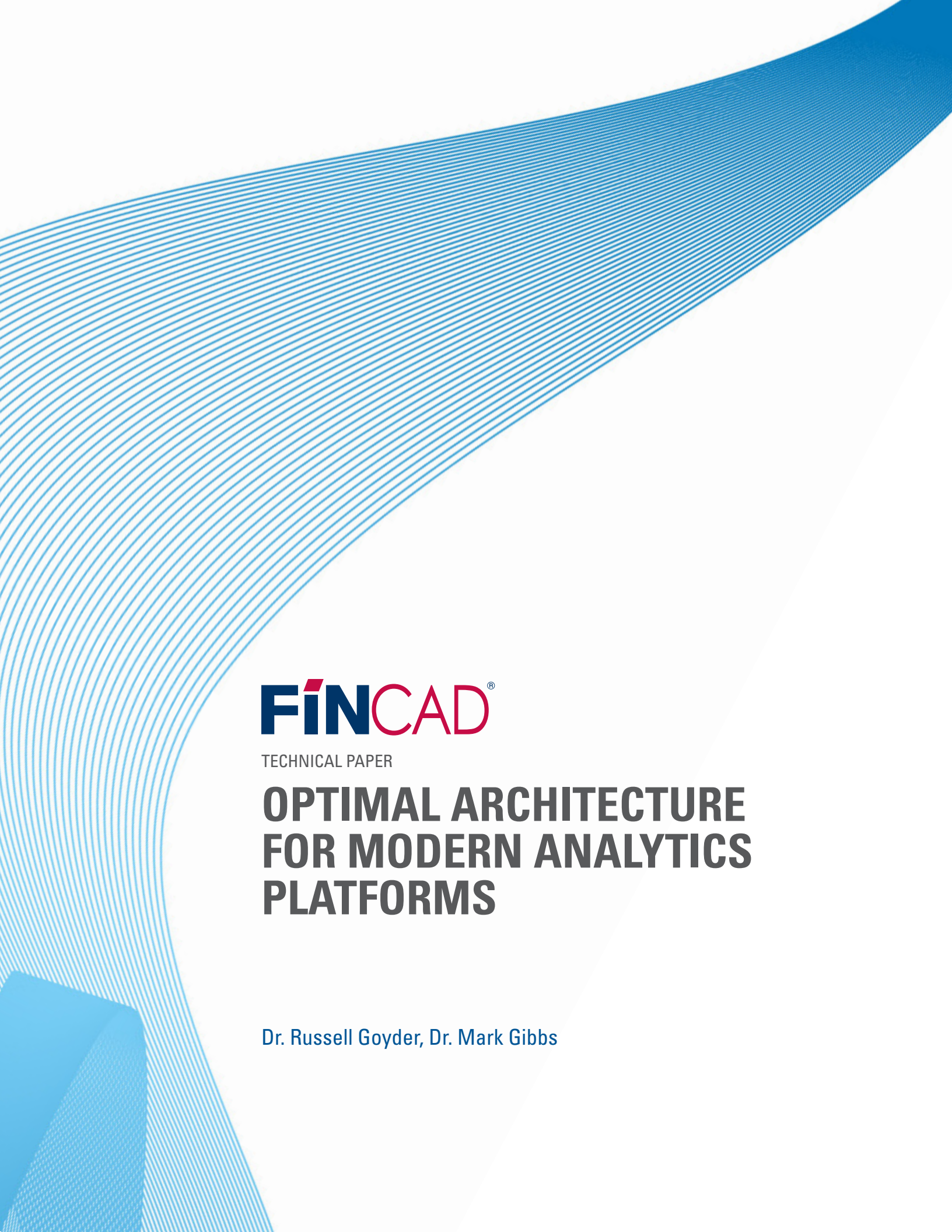

FINCAD®

TECHNICAL PAPER

OPTIMAL ARCHITECTURE FOR MODERN ANALYTICS PLATFORMS

Dr. Russell Goyder, Dr. Mark Gibbs

Optimal Architecture for Modern Analytics Platforms

1 INTRODUCTION

Financial derivatives are now in ubiquitous use around the globe to hedge exposure and as vehicles for speculation. Four years on from the worst financial crisis in almost a century, the world is still recovering. Yet the derivatives markets continue to grow, and have more than recovered from the slight pull-back after the crisis of 2008. According to the *Bank of International Settlements*, as of June 2011 the total global notional of outstanding OTC derivatives was US$707 trillion, having surpassed the previous high point of around US$670 trillion in June 2008. Prior to that, the market had been doubling every three years or so. The fundamental need to transfer risk in exchange for long-term reward will persist, while financial markets throughout the world continue to develop in size and maturity.

Prior to the crisis, not only had the volume of trading in derivatives increased massively, but so had their complexity. There is no clear sign yet what the long term trend will be going forward. On one hand, investors seek more aggressive and sophisticated ways to save for retirement using structured products, as the effects of demographics, low interest rates, and low prior savings rates come home to roost. On the other hand, there is increasing regulatory pressure to use simpler and more transparent contracts, loss of trust, and concerns about liquidity of complex securities. Sell-side banks seek higher margins by creating structures that accommodate bespoke exposure profiles to manage clients' specific risk – complex trades require more sophisticated pricing and such knowledge commands a premium, yet higher capital requirements will offset the profitability.

Regardless of the uncertain trends in complex trades, increased regulatory scrutiny and a heightened sensitivity to counterparty credit risk since the crisis have forced fundamental changes in how even the most vanilla derivatives are valued. More widespread use of collateral has imposed new levels of complexity on the most basic and preliminary task for any financial calculation: the construction of a discount curve. The option to choose collateral currency, nominally present in some collateral agreements, has even prompted analysis of whether a so-collateralized vanilla swap in fact has non-negligible exposure to the volatility of certain spreads. Accounting for the effect of counterparty exposure (calculating a Credit Value Adjustment - CVA) on the value of a portfolio of vanilla derivatives requires a level of sophistication matching that previously applied to highly complex exotics. Even without CVA, certain types of life-insurance policies available to retail investors cannot be valued appropriately without sophisticated modeling and computational techniques traditionally only needed for exotic trades.

The modeling of financial derivatives is performed by quantitative analysts, often with backgrounds in disciplines such as physics and mathematics, well-versed in stochastic calculus and arbitrage-free pricing theory popularized by Black, Scholes and Merton [1], and formalized by Harrison and Pliska [2]. Their critical contribution is in making well-judged modeling decisions that provide adequate account for the factors that influence the value of a trade or portfolio.

A practical reality of a quant's role is that calculations must be implemented in software systems. This is a somewhat exacting requirement; software engineering itself is a craft to which many devote their full careers. Nevertheless, quants are typically proficient in the techniques necessary to implement numerical computations that result in accurate and robust numerical results. A general consequence of this emphasis of software competence toward the details of a numerical simulation is that wheels are reinvented frequently in many parts of many institutions. New variants of existing pricing calculations are implemented without direct reuse of the shared parts. New trades or models prompt bespoke development which at times amount to nothing more than a highly paid form of repetitive manual labor.

The missing ingredient here is architecture; the overall design of a system informed by a deep understanding of the fundamental concepts which underpin the domain and the nature of the information that flows and interacts when problems are solved in the domain. It is architecture that facilitates true modularity and sub-component reuse, and without a clear treatment of the concepts relevant to the problem domain, the goal of constructing an effective architecture remains elusive. A major cause of this deficiency is that individuals with the necessary rich architectural vision are rare and expensive, particularly outside the realm of top-tier sell-side institutions.

In this paper we tackle precisely this issue; the fundamental concepts and design ideas that must underpin the architecture of a modern analytics platform. Our focus is not modeling, but the framework within which models function. The primary conceptual distinctions follow from considering closely the characteristics shared by all financial derivatives contracts, which is done in *Sec. 2*. Modeling is discussed in *Sec. 3*, but briefly, citing other sources for a more detailed treatment. In *Sec. 4* an example is given of an embodiment of the ideas introduced in this paper and in *Sec. 5* several pragmatic issues are addressed which are not specific to the domain of financial derivatives but relevant to a far broader collection of problem domains. Finally, we make concluding remarks in *Sec. 6*.

2 GENERIC TRADE DESCRIPTION

Let us consider financial derivatives as a whole and develop a conceptual framework in which any such derivative can be described. In doing so, we will focus on the information content of the relevant legal documents that is pertinent to the problem of calculating the derivative's value. An important subsequent step is the development of the conceptual structure of the information that, while necessary for valuation, is not present in the contract. This leads us to our first and most important conceptual distinction that must underpin the design of a modern derivatives valuation platform.

2.1 The Product concept

A financial derivative is a legal agreement between a collection of (usually two) parties. The terms of the agreement are expressed in a document, the term sheet. As with any legal document, a term sheet expresses obligations and rights. The obligations relevant to the question of how much one party might be willing to pay for the privilege of entering such a contract, are those to make payments. Such payments might be of cash or of some other asset. The promise to pay, say, one dollar in a year's time is perhaps the very simplest example of such a contract.

The rights relevant to a contract's value are those to choose one course of action from a collection of many. The only aspects of the available courses of action relevant to the question of how much such a right is worth, are the obligations to make payments (and rights to make subsequent choices) contained within them. However, we have already associated the notion of "financial derivative contract" with such a collection of payment obligations and rights. It is clear that a recursive structure is emerging; we can model the valuation-specific aspects of a derivative contract by defining the concept of a *Product* as a collection of

Definition 1. The Product concept

- obligations to make payments of cash or other assets and
- rights to other Products

(the name *Product* reflects the sell-side origin of the ideas developed in this paper). For example, a Bermudan swaption represents a series of choices, each on a prescribed date. When the first choice arrives, the holder of such a contract has the right to enter an interest rate swap with a counterparty. The holder may choose not to enter such a contract at that time, in which case another opportunity to choose will arrive. Typically, such subsequent choices offer entry into the remainder of the same underlying swap, but this is not a requirement. We can see several nested *Products* here, illustrated in *Fig. 1*.

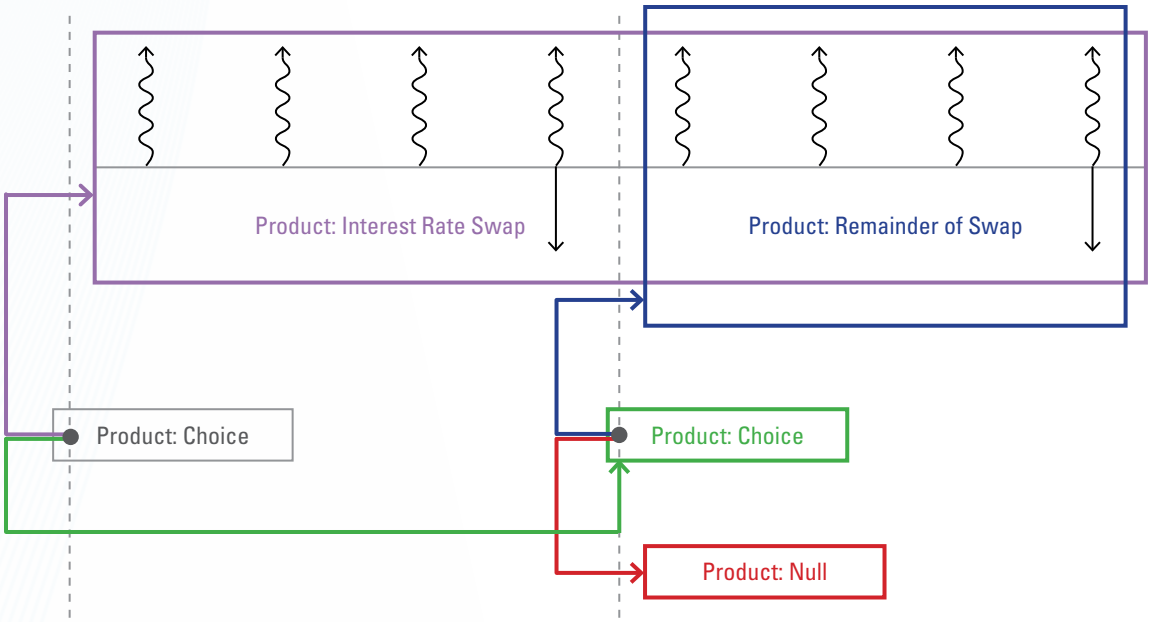

Fig. 1. Schematic illustration of the Product structure of a Bermudan swaption

The Bermudan swaption itself is a *Product*, equivalent to the first choice, because all other relevant *Products* are contained within the two underlyings of that choice. One branch of the choice leads to an interest rate swap, another *Product,* but one consisting of only payment obligations, no rights to make choices. The other branch of the choice leads to a subsequent choice between the remainder of the same swap (another *Product*) and, because in this example the second choice is the last, nothing. Strictly, a contract devoid of any obligations or rights altogether is yet another *Product*, in the same way that zero is a number.

2.2 Product types

The only essential content of a *Product* is the list of payment obligations and rights to make choices which it assigns. However, it is convenient to add further structure to this conceptual framework by identifying a number of types of *Product* based on their content.

- **Flow**
  A Flow consists of the obligation to make a single payment, without any choices. Information required to uniquely characterize (the value of) such a Flow includes the following.
  - **Notional**
    The topic of specifying the amount of a payment is covered in detail in *Subsec. 2.3*. However, it is common practice to specify in addition to the ideas developed in that section, an overall multiplier, the notional.
  - **Direction**
    From the point of view of one party to a contract, payments (that contribute to the contracts value to that party) are either made or received. A sign convention must be chosen, with the natural one being that a positive sign indicates receipt. Some *Product* structures are highly complex, with many levels of nesting of *Products*, so expressing payment direction with a naked negative sign is highly error prone. It is safer for each *Product* to be associated with an explicit instruction to pay or receive, although during any valuation, the effect is to insert minus signs in appropriate places.

- **Currency**
  Cash payments must be made in a currency. When the value of a *Product* is reported, it is natural to do so as a collection of values in each currency in which payments are made. Calculating the value of a multi-currency *Product* in a single currency is a separate step and part of the details of a particular valuation approach rather than being anything intrinsic to a *Product*. Indeed, the reporting currency for a given valuation need not be any of those in which payments are made.
- **Auxiliary information**
  Given that this paper considers the problem of developing a platform for the valuation of derivatives, it is natural to divide the total information content of a particular *Product* into two categories: that pertaining to its value and everything else. While not essential to valuation, it can be useful for a Flow to be labeled with auxiliary information, such as the counterparty, an ID indicating to which trading book the Flow belongs, or the ID of the Flow in some external system.

- **Leg**
  A Leg consists of a collection of Flows, all made in the same direction and in the same currency. It is the generalization of a single Flow to the idea of Flows made on a schedule. Such a schedule may be generated from functionality within the library, but schedule generation is a separate problem and the construction process for an object representing a Leg should consume a schedule, not instructions for generating one. A further concept that is most readily associated with a Leg is that of a notional structure; a collection of notional amounts, one per Flow, that varies in time.
- **Swap**
  A Swap consists of two Legs, with payments made in opposite directions. Swaps are very common structures and so having an explicit concept for them is of great utility. It facilitates the calculation of quantities specific to swaps such as a par rate or margin. Note that this concept covers all types of swaps, from vanilla interest rate swaps or basis swaps to credit default swaps, cross-currency swaps or swaps with exotic payoffs. The distinguishing feature of these different types of Swaps is the nature of the amounts that are paid in each constituent Flow. The definition of payment amounts is covered in detail in *Subsec. 2.3*.
- **Portfolio**
  A Portfolio consists of a collection of *Products*. Note that a Portfolio is therefore also a *Product* and should have no special status in a valuation system. It should be valued on the same footing as all other *Products* and every calculation that works for an individual *Product* should also support Portfolios. A Portfolio of a collection of Flows is equivalent to the corresponding Leg. A Portfolio of two offsetting Legs is equivalent to the corresponding Swap.
- **Choice**
  A Choice contains no payment obligations, only the right to make a single choice between two other *Products*. The payment direction indicator, introduced above in the context of a Flow, here indicates whether a party or their counterparty buys or sells the right to choose. A further Boolean value must indicate whether a given party chooses a given *Product* for themselves or for their counterparty to own. From the point of view of valuation, and assuming rational parties, this amounts to whether the highest or lowest value branch is selected. A choice between more than two alternatives can be constructed from the binary Choice described here, and so does not merit treatment as a separate *Product* type.

- **Conditional**
  A Conditional obliges a party to a contract to take ownership of one *Product* or another depending on some condition. The condition would be based on some quantities whose values can be observed unambiguously and agreed upon by all parties to the contract. This notion is developed in detail in *Subsec. 2.3*.

We have covered some useful subconcepts within the idea of a *Product*, but this is far from an exhaustive list. It is possible to construct higher-level structures such as that of the Bermudan swaption in *Fig. 1*, but this just amounts to further convenience; each such structure conforms to the definition of a *Product* and can be constructed using the above building blocks.

Throughout this section we have made the tacit assumption that Flows are measured in units of currency, not some other asset. However, everything we have said applies equally well to cash-settled and physically settled trades alike. There is one area, however where the distinction matters. From the point of view of valuation, the obligation to make a cash payment has value as long as it has not yet occurred. Flows made in the past do not contribute to the value of a derivative. In order to continue to reflect the value of cash received in a given Flow, we would, from the point of view of a derivative valuation system, invest that cash in another vehicle such as a cash deposit where our counterparty is obliged to make a cash payment to us in the future.

This contrasts sharply with payments of physical assets such as bonds, where the receipt of such assets reflects their ongoing ownership. For example, a bond trade is best described as a Swap consisting of two Flows, one cash settled (the agreed price of the bond) and the other physically settled (representing receipt of the bond itself). Before the trade is settled, its value is the present value of the difference between the forward price of the bond and the fixed payment to be made for it. After settlement, the value is that of the then market price of the bond.

2.3 The Index concept

So far, we have described how parties to a contract may be obliged to make payments, but we have said nothing further about the nature of those payments. The key requirement for the definition of a payment is *unambiguous determination* of the amount to be paid, when required. Note that when the payment amount is determined and when the payment is made are two distinct events, with the latter falling no earlier than the former. Indeed, a small delay is common, to accommodate the practicalities of delivery. An indicative list of those unambiguously determinable quantities which form the definition of payment amounts would include

- equity prices,
- equity indices,
- interest rates,
- whether a credit entity has defaulted, and
- foreign exchange rates.

However, this is very far from an exhaustive list. It is these quantities which are the “underlyings” from which a derivative contract is derived and therefore is so-called. The common feature of all of these quantities is their ability to form part of the definition of a future payment in a contract written today. For each it is possible to refer to an established methodology for observing the value in the future. Sufficient detail must be included, such as the exact definition of credit default for example, and if there is an appreciable chance of an observation not being possible when required, further provision must be made for an alternative course of action. Such details are the province of the legal counsel of each party to a contract; for our purposes is it sufficient

to recognize that there is a concept, that of an *Index*, which can be characterized as

Definition 2. The Index concept

- the definition of a quantity on which a payment is contingent.

It is critical to note the careful terminology in the above statement. The *Index* concept concerns the *definition* only of an underlying, not its actual value at any given time. For example, the British Bankers' Association provides the definition of several *LIBOR* rates on its web site. It is possible to read how LIBOR values are obtained (it is by topping and tailing the results of an opinion poll of 16 London banks). The same organization also documents the rules and conventions for determining the start and end dates of the borrowing period relevant to each rate, given an observation (or fixing) day. The inverse rules are also documented, showing how to determine the correct fixing date for a given borrowing period. All of this information is part of the definition of the concept of LIBOR and all of it pertains to every LIBOR rate ever observed.

The *Index* concept makes this subtle distinction between the definition of the value and the value itself, because its purpose is to encode the information present in the legal contract, where it is sufficient to describe how, at some future time when an observation of the underlying is required, to go about making the observation.

2.4 Complex payoffs

Derivative contracts often oblige parties to make payments that depend on a collection of underlyings in complex ways. For example, a contract might instruct that a payment is made on some date d whose amount is given by

- the greater of zero and the Total Capped Return as observed on date v,
- where the Total Capped Return is the smaller of 20% and the Combined Capped Return,
- where the Combined Capped Return is the sum of the Capped Return for each of three stocks, A, B and C,
- where the Capped Return for stock X is defined as the smaller of 10% and the difference between the ratio of the value of stock X on some date v to that on date u, and one.

This style is typically found in term sheets, but with more legal rigor, omitted here for clarity. Increasingly, often mathematical notation is used directly in term sheets, which results in a more compact expression of the information content, as follows

$$\max(0, \min(\min(R(A), 10\%) + \min(R(B), 10\%) + \min(R(C), 10\%), 20\%)) \tag{1}$$

where

$$R(X) = \frac{X(v)}{X(u)} - 1 \tag{2}$$

for some underlying X. Our concern is the efficient encoding of this formation for use in derivative valuation platforms. The key observation is that, if A is an *Index* that holds the definition of the value of shares in company A, then $R(A)$ also meets the requirements of being an *Index*. The introduction of mathematical or computational operations or functions (which are certainly well-defined and unambiguous) such as arithmetic, logic, or calculations such as $\max$ and $\min$, results in the definition of a quantity that is just as well-defined and unambiguous as the formula's inputs, and therefore is just as capable of being written into a financial derivative contract.

Although *Eq. (1)* may seem complex, the number of distinct operations which are used to express payoffs is not large. There is a fundamental set which can accommodate virtually any payoff. This set includes the following constructs:

- Arithmetic operators +, -, * and /
- Boolean operators <, >, <=, >=, **and**, **or** and **not()**
- Conditionals; **if(Condition,Left,Right)**, evaluates to **Left** if **Condition** is true, otherwise **Right**, where **Condition** is a Boolean-valued *Index*
- The **min** and **max** functions, together with other special functions such as **logarithms**

These are all *low-level* constructs; they are fundamental operations from which others may be composed. For example, we could synthesize a new payoff called call via **call(X, k) = max(X - k, 0)** for some strike **k** and underlying **X**. This is of course just a European call option payoff. Similarly, other higher-level Indices may be constructed, such as an *Index* whose value is the fraction of time that a condition is met during an interval of time (which would cover all range accrual payoffs) or an *Index* which indicates whether some underlying *Index* breached a barrier or set of barriers.

When encoding payoffs in a derivatives valuation platform, it is most convenient to construct a *language* in which payoffs may be expressed. Such a language should contain all of the above fundamental constructs, but there are two distinct advantages to including higher-level expressions also. The first is operational efficiency; high-level expression of intent is desirable in any programming language, because it enables a concise expression of the programmer's instructions, which in turn results in code that is cheaper to maintain. The second is more subtle. By retaining the concept of "barrier" or "range accrual" or similar, instead of expressing the equivalent payoff calculation in terms of fundamental operations, it is far easier to optimize the calculation of such payoffs with bespoke algorithms or approximations. Without retaining such concepts, we would be forced into a challenging pattern recognition exercise within the calculation tree.

2.5 Modeling and valuation

In *Subsec. 2.1* we defined the concept of a *Product* as a collection of obligations to make payments (and rights to other *Products*). We were careful to delegate the task of stating the payment amount to another concept, that of an *Index*, introduced in *Subsec. 2.3*. We saw in *Subsec. 2.4* how the *Index* concept was far more general than the simple examples (underlyings) used to introduce it and in fact covers payoffs of arbitrary complexity.

However, at no point in the above have we mentioned *modeling*. We have alluded to it, in so far as we have focused on the information content within a contract that is pertinent to its financial worth, but we have stayed within the realm of the lawyer and not strayed into the mathematical domain of the quantitative analyst. This may seem surprising at first sight, because *Eq. (1)* certainly does appear mathematical, and indeed it is, but it is strictly part of a legally binding term. It influences the value of the contract in the same way that the notional does; it determines the actual cash amount that is settled on date . On this date, no modeling assumptions are necessary, the value of each underlying is known, and a simple calculation will render the amount that one party owes another.

The task of the quantitative analyst (in the application of conventional risk-neutral pricing), is to form a suitable *prediction* of these payment amounts ahead of time. In other words, to make a well-chosen set of modeling assumptions for the relevant underlyings that allow the construction of a probability distribution of present value of the contract, whose mean is the arbitrage-free

price. This is now the world of sophisticated mathematical and computational techniques, and modeling judgment.

Identifying these two realms as distinct, and defining our concepts accordingly, is a very deliberate choice, and a subtle one. Historically, due to the mathematical nature of both complex payoffs and modeling techniques, they were often blended together in the design (or evolution) of software systems. If we instead keep what is valued (*Product*, *Index*) separate from how it is valued (modeling), then there is far greater opportunity for reuse of subcomponents. Exotics and vanillas are valued in a consistent manner. The same modeling code can be used to value multiple contracts, and a given payoff calculation can be reused for a variety of modeling assumptions. Modeling effort is expended once, not on a per-trade basis customized for each payoff. And by being clear on where modeling ends and payoff expression takes over, each payoff component is written just once, for all models.

The separation also aids conceptual clarity. When comparing the value of a single contract under N different modeling assumptions, it is very useful to have a single object representing the contract and N objects for the models. Furthermore, within a given set of modeling assumptions for a group of underlyings, the task of pricing new structures does not need the expensive mathematical and computational sophistication of a quant; only an understanding of the simpler constructs found within financial payoffs and expressed by means of *Indices*. The ultimate goal of the providers of pricing libraries is to scan a term-sheet automatically (or to facilitate the representation of the legal agreement electronically in the first place). The conceptual separation of *Product* from modeling is an important step toward this goal.

3 GENERIC MODELING

We now turn to the topic of modeling and its associated conceptual structure. The most obvious distinction to make in this context is that between idealized mathematics and the practicalities of implementation of a calculation as a numerical computation. This is certainly a valid separation - the concept of the Black model [1] exists quite independently of whether the associated stochastic differential equation is being solved in closed-form, via its characteristic function or in a Monte Carlo simulation. However, experience has shown that an abstraction for the idea of a specific idealized mathematical model is of limited practical utility; that a subtly different conceptual separation lends itself more readily to derivative valuation. The two associated concepts are *Model* and *ValSpec*.

3.1 The Model concept

The idea of a *Model* represents

Definition 3. The Model concept

- a consistent, arbitrage-free view of the market in which a given collection of Products is to be traded and hedged, at the time of valuation.

Whenever a parameter from a given choice of mathematical models is required during a valuation, it is the *Model*'s role to provide it. Each different *Model* represents a distinct collection of modeling assumptions used in valuation. Note the capitalization of the M in *Model* when referring to the concept defined in *Definition 3*. Typical *Models* contain hundreds or thousands of such parameters, but for the purposes of illustration, we will invent a highly stylized toy *Model*, which might consist of the following information:

- the time value of money in USD is modeled as an annual discount rate of 50bps
- the expected value of 3-month USD LIBOR is given by a 100bps spread over the 3-month forward rate implicit in the above discount curve
- the expected value of 6-month USD LIBOR is given by a 50bps spread over 3-month USD LIBOR
- where the expectations are conditional on all the information available at the current time

This simplistic *Model* will yield a value for vanilla interest rate swaps and 3-6 month LIBOR basis swaps in the USD market, but will not for Flows in any other currency and for any *Products* whose underlyings are equities, credit entities, bonds, interest rates of other tenors or any other underlying outside the set of 3 and 6-month USD LIBOR (paid close to the end of the corresponding period, so that no convexity adjustment, and therefore volatility information, is required). For an example of the level of sophistication required to build a *Model* consistent with even the vanilla rates market in a collection of currencies in the current post-crisis landscape, see [3].

We can see from the above example that artifacts in a *Model* must be present for each underlying in the *Index* expressions specifying the payment amounts in a given *Product*. If a *Product* explicitly specifies a foreign exchange rate, then that is such an underlying and its spot and forward values must be in the *Model*. If a given *Product* does not reference an exchange rate explicitly, but does contain Flows in multiple currencies, then a valuation instruction to report the value in a single currency may require the presence of the relevant exchange rates. We see therefore that the set of required *Model* artifacts is determined by both what is being valued (the

Product) and how the valuation has been instructed to proceed. Such valuation instructions are covered in *Subsec. 3.2*.

In addition to the quantity of underlyings and valuation instructions which our simple *Model* can accommodate, the quality of the modeling leaves a lot to be desired. In particular, the somewhat arbitrary assumption of an annually compounded discount rate is unlikely to be consistent with any real market. Market consistency is achieved by means of *calibration*; the act of tuning model parameters such that the prices of market instruments match quoted values.

3.1.1 Efficiency

It is clear that a *Model* must contain a snapshot of all relevant market quotes, to which model parameters are to be calibrated. By choosing to have a single object with a global view of the information pertaining to a particular market, it is possible for that object to manage the calibrations efficiently, by tracking the relationships between model parameters and market data. In our simple example above, if we change the discount rate from 50bps to 60bps, all three curves are affected. If on the other hand, we decide to model 3-month USD LIBOR with a spread of 90bps not 100bps, then both the LIBOR curves are affected while the underlying discount curve is not.

Although trivial in the simple example above, realistic *Models* contain complex relationships between model parameters and market quotes, and often run expensive calibrations. An automatic manager of those relationships, that keeps track of which calibrations are and are not affected by changes in underlying quotes or modeling assumptions, saves considerable effort and expense.

We have seen that a *Model* tracks which calibrations are invalidated when it is updated, but we have not addressed the question of when calibrations are run. Given the *definition* of the *Model* concept, we can see that it is intended to cover the entire set of underlyings traded within a given market. Given the size of most markets, one could be forgiven for thinking that the construction of such a *Model* could take a prohibitively long time.

The solution to this problem is to make *Models* pay-as-you-go. No calibrations are run when a *Model* is constructed. Rather, calibration instructions are stored for later use. It is only when a given model parameter or curve is requested by a particular valuation that calibrations are run and curves are built. Furthermore, when a curve (say C) is built, it is saved for use by any subsequent valuations. It is only when another curve or quote on which C depends is updated, or the calibration instructions for C are changed, that the built curve is invalidated. When a curve is invalidated, it is not simply deleted. Instead it is saved for use, if desired, as the starting point for a subsequent calibration. Typical market moves are not large and so this can result in considerable reductions in computational effort over a less well-informed initial condition.

Given that a *Model* covers the complete set of underlyings traded in a given market, and that underlyings are referenced explicitly by means of *Indices* in *Products*, the associated valuation machinery can choose which curves and other model parameters are to be requested from the *Model* automatically. There is no need for a user to pass curves explicitly into any valuation call. Even a *Product* as simple as a vanilla interest rate swap paying 6-month LIBOR with a 6-week stub, where contracts specify interpolation between LIBORs of adjacent tenors, would query the *Model* for its 1, 2 and 6-month LIBOR curves. To value a portfolio of swaps with stubs of varying lengths, and to combine these swaps with other asset classes, would quickly result in a combinatoric explosion of curves. In contrast, the *Model* concept accommodates this complexity (and far more) with little effort.

3.1.2 Arbitrage and consistency

Part of the *definition* of the *Model* concept states that a *Model* affords an arbitrage-free view of a given market. This is achieved by enforcing that there is only one version of each curve present in a *Model*; one modeling assumption for the time-value of money in each currency, one forward curve for each asset, one modeling choice for each forward rate curve and so on. If a *Model* contains quoted exchange rates for the currency pairs A:B and B:C, it will not accept a quote for A:C, because that cross rate is already implied from its existing market data and allowing a separate, potentially inconsistent quote, would admit arbitrage opportunities. Of course, if the user desires to use the quote for the A:C pair, she is free to do so, but that is a distinct *Model*, which can be compared to the original by simply valuing a *Product* in the context of each one.

So far we have discussed curves; expected *Index* values given information known today (at the time of valuation), calibrated to quotes for instruments for which static hedges are possible. *Products* containing Choices require future expectations to be evaluated; they require the dynamics of underlyings to be calculated. A single *Model* can contain multiple specifications of the stochastic processes by which an underlying is assumed to evolve, but the static part of such a model must be common to all such processes. This again is because it is only by allowing multiple inconsistent assumptions in a single *Model* for the static components (forward curves, for example) of a given underlying's evolution that arbitrage opportunities can arise in a valuation. The task of selecting which stochastic process a given underlying is assumed to follow is not part of the *Model*, and is addressed in *Subsec. 3.2*. For the current discussion of *Models*, it is sufficient to note that only one stochastic process can be associated with a single underlying within one valuation.

3.2 The Valuation Specification concept

We have seen how a *Product*, together with its *Indices*, encodes the information necessary to perform valuation. Modeling choices for the observables underlying a given *Product* are made in the *Model*, which manages calibration efficiently and provides a consistent view of the relevant market. In this section we explore the concept of a *Valuation Specification*, or *ValSpec* for short.

A *ValSpec* is defined to be

Definition 4. The ValSpec concept

- how valuation is to proceed, numerically.

The *Product* represents what is valued and the *Model* can be thought of as representing where (in parameter space) the valuation occurs. The *ValSpec* represents how we achieve the practical outcome of a numerical result being emitted by a machine. To illustrate, the following items of information would be contained within a *ValSpec*:

- a method for calculating a common measure of time, given date information,
- the currency in which the value of a trade is to be reported, if such a request is made,
- the approximation method for calculating the integrals needed to value a CDS,
- whether to include cash flows on the valuation date in the valuation,
- which underlyings (interest rates, equities, FX rates etc) to simulate via Monte Carlo and which stochastic process each should follow,
- for a Monte Carlo simulation, how many threads to use (or how to access a grid or cloud), the stopping condition and the underlying random number generator,
- the algorithm for calculating the weights of European swaptions in the static portfolio replication of a CMS payoff and
- an instruction to evolve backward in time via the repeated application of characteristic functions.

The above information has some common features. It is all independent of the legal contract being valued and is not specific to a particular curve or model parameter. It is all concerned with the numerical algorithms, approximations and other practical configuration details that are needed for a valuation. As the main focus of this paper is the approach to valuing a generic trade, we next consider a Monte Carlo approach to valuation, because it is the most general numerical method available to the quantitative analyst.

3.3 Generic Monte Carlo simulation

A full examination of the design considerations that go into a generic Monte Carlo engine is beyond the scope of this paper. Instead, we give a brief overview of the high-level requirements for such a capability and leave a detailed exposition to a forthcoming publication [4].

- **Hybrid modeling**
  For multiple underlyings, the joint distribution of all underlyings being simulated can be a complicated function. A good Monte Carlo engine will allow the user to specify the stochastic process for the marginal distribution for each underlying individually, together with a correlation structure, and calculate the joint distribution without further input. In particular, each marginal distribution may be defined in a measure that is not the risk-neutral measure for the simulation, but the Monte Carlo framework should correct for such effects automatically. Furthermore, any stochastic process should be available to model any underlying asset and any term-structure model should be available to model any underlying rate.
- **Multithreading and distributed computing**
  The simulation should make full use of multi-core machines for all calculations and have the capability to distribute all calculations across multiple computers. It should be possible to specify these choices conveniently and concisely. An option to automatically detect the number of cores available on a given machine and use them all (or perhaps all but one) would be an added extra.
- **Full analytic risk for all valuations**
  With Monte Carlo simulation a bump-and-grind (finite difference based on perturbing inputs) approach to evaluating the sensitivity of price to changes in the quotes to which model parameters are calibrated is prohibitively expensive. It is therefore desirable to have the partial derivative of the value of an arbitrary *Product*, V with respect to each market quote q_i calculated alongside the value itself

  $$V, \frac{\partial V}{\partial q_i} \quad \forall i = 1 \ldots N$$

  for N quotes in total. The cost of a bump-and-grind approach is linear in N, whereas an analytic approach to the calculation is essentially constant with respect to the number of exposures, and so they all should be calculated. For details of an analytics platform which guarantees the ability to calculate full analytics risk for all *Products*, see [5].
- **Arbitrary choice handling**
  No Monte Carlo framework is truly generic unless it possesses the capability to value *Products* that contain an arbitrary structure of (perhaps nested) choices [4].

- **Automatic simulation generation**
  There is sufficient information within a *Product* to determine the collection of times at which the *Indices* underlying its payoffs must be observed in order to calculate its value. In addition, the mathematical operations within the payoffs themselves map directly to calculations that operate on the simulation variates. Modern compiler technology allows a *Product* to be parsed and the relevant Monte Carlo simulation to calculate value and full analytic risk to be constructed on-the-fly. There should no longer be any need for a human to code a simulation by hand.

  The only information that should be necessary to supply to calculate the value of a *Product* by simulation is
  - the *Product* itself,
  - a *Model*, configured with the necessary model parameter calibrations,
  - a choice of stochastic process for each underlying, defining its marginal distribution and
  - simulation details like the stopping condition, number of threads and the base random number generator.

For a more in-depth analysis of the topics touched here briefly, we refer the reader to [4]. In the remainder of this paper, we examine an embodiment of the design principles described above applied to solving some common problems within quantitative finance.

4 VALUATION EXAMPLE IN F3

F3 is a modern analytics platform whose architecture represents a distillation of the accumulated wisdom of the past three decades of sell-side analytics library development. In this section, we examine the simple act of valuation in F3 which clearly embodies the design concepts presented in this paper. We then use F3's generic valuation capabilities to structure and price a variable annuity (*Subsec. 4.2*) and calculate its exposure to all market quotes.

4.1 The act of valuation

The act of valuation in F3 is performed by the function ValueProduct, shown using *Microsoft Excel* in *Fig. 2*. We are free to choose Excel because F3's API is consistent across all calling environments. In *Subsec. 4.2* we choose *MATLAB* because that environment lends itself more readily to the visual presentation of the F3 function calls being made.

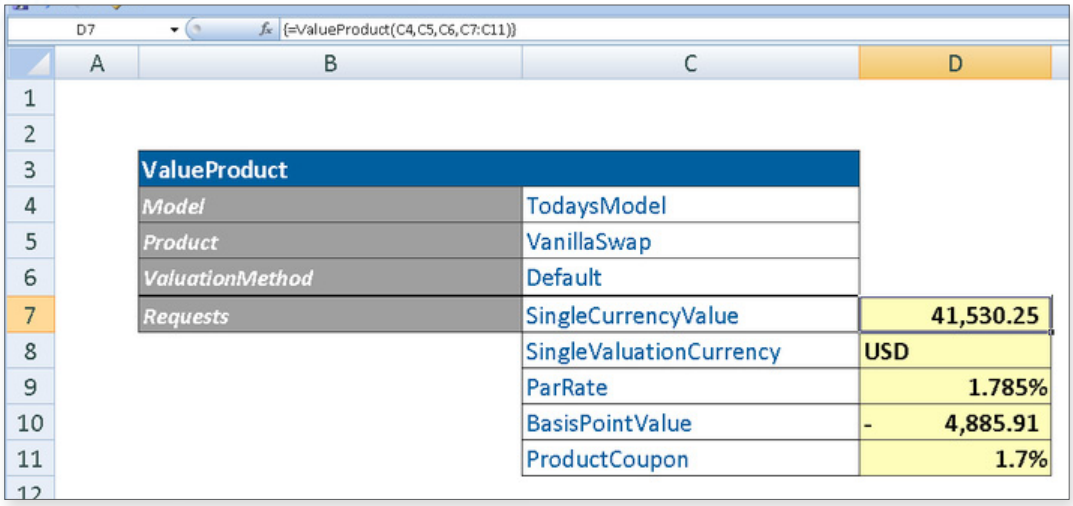

Fig. 2. Screenshot of an interest rate swap being valued in F3 Excel Edition.

The formula (function call) is in the yellow cells in column D, while the name of the function being called is shown in the blue row at the top (row 3). The names of the four input arguments are shown in the gray cells in column B while their values are in column C (white cells with blue text).

- **Model**: As input, we supply the name of a *Model* object, representing the concept defined in *Definition 3*. Modeling choices are not the main focus of this paper, so here we simply assume that the given *Model* contains enough information to price the given *Product*. In the case of the vanilla swap whose valuation is shown in *Fig. 2*, the valuation would require that the *Model* contain a discount curve for U.S. dollars and a curve giving the expected value of the rates paid by the floating leg of the swap.
- **Product**: We supply the name of a *Product* object, representing the concept defined in *Definition 1*. In this example, a vanilla interest rate swap is being priced. In F3, such a swap can be structured explicitly in terms of the Flow, Leg and Swap *Product* types, with the relevant *Index* for the floating rate, or we can ask F3 to do it for us by calling the function *CreateInterestRateSwap*.
- **ValuationMethod**: Here we supply the name of a *ValSpec* object, representing the concept defined in *Definition 4*. In this example, we only require default settings for a closed-form calculation. So we do not need to construct a *ValSpec* object explicitly. F3 has several objects built-in, including a default *ValSpec*.

- **Requests**: This input argument expects a list of requests that specify the desired output from the valuation. In this example, we have asked for the value of the swap, and its par rate, which we can compare with the fixed rate being paid by the swap in order to understand why the value is not par. In other words, since inception of the swap, the market has moved such that the swap no longer values to par.

4.2 Variable annuity

The term *variable annuity* describes a popular class of investment vehicle offered by insurance companies to the retail market where the policyholder invests an amount in an equity vehicle for many years (the accumulation phase) which is then converted to an annuity (typically at retirement age) whose payments depend on the value of the equity investment. Often, embedded options providing various forms of safety can be found, which command an additional premium. In this example, we structure and value a flavor of a variable annuity known as a Guaranteed Minimum Income Benefit (GMIB), as described in [6] and references cited therein.

Our focus will be on the *Product* structure, not the modeling assumptions that go into a particular valuation. For this example, we will assume that a *Model* is available that models the dynamics of the relevant equities, interest rates and policyholder survival rates, together with their correlation. A common simplifying assumption is that the correlation of policyholder survival with market factors can be safely neglected. We can then use actuarial tables directly to obtain the necessary survival probabilities. While modeling assumptions are kept in the background, the market quotes from which model parameters are calibrated will be visible when we analyze the exposure of the GMIB. Modeling and *Models* will be the topic of a separate paper, [7].

4.2.1 Term sheet

The term sheet for the GMIB considered here can be summarized as follows:

- the policyholder invests an amount N at time t_0 in a fund whose unit price is $S(t_0)$.
- at some later time T, the policyholder has the right to choose whether to receive a single payout of

$$N\frac{S(T)}{S(t_0)}, \tag{3}$$

the then value of the investment in the fund, or an annuity which pays each year at time until the policyholder's death an amount given by

$$Ng\max\left(X(T), Y(T)\right) \tag{4}$$

where g is the *annual payment rate*, $X(T)$ is the *guaranteed minimum fund level*, expressed in terms of a *guaranteed rate* r_g

$$X(T) = (1 + r_g)^{T-t_0} \tag{5}$$

and $Y(T)$ is the fund's peak before T

$$Y(T) = \max_n \left(\frac{S(t_n)}{S(t_0)}\right) \qquad n = 1, 2, \ldots, T. \tag{6}$$

The relevant underlyings for valuing this policy are the value of the equity fund, U.S. dollar interest rates and the survival of the policyholder. This structure is illustrated in *Fig. 3*.

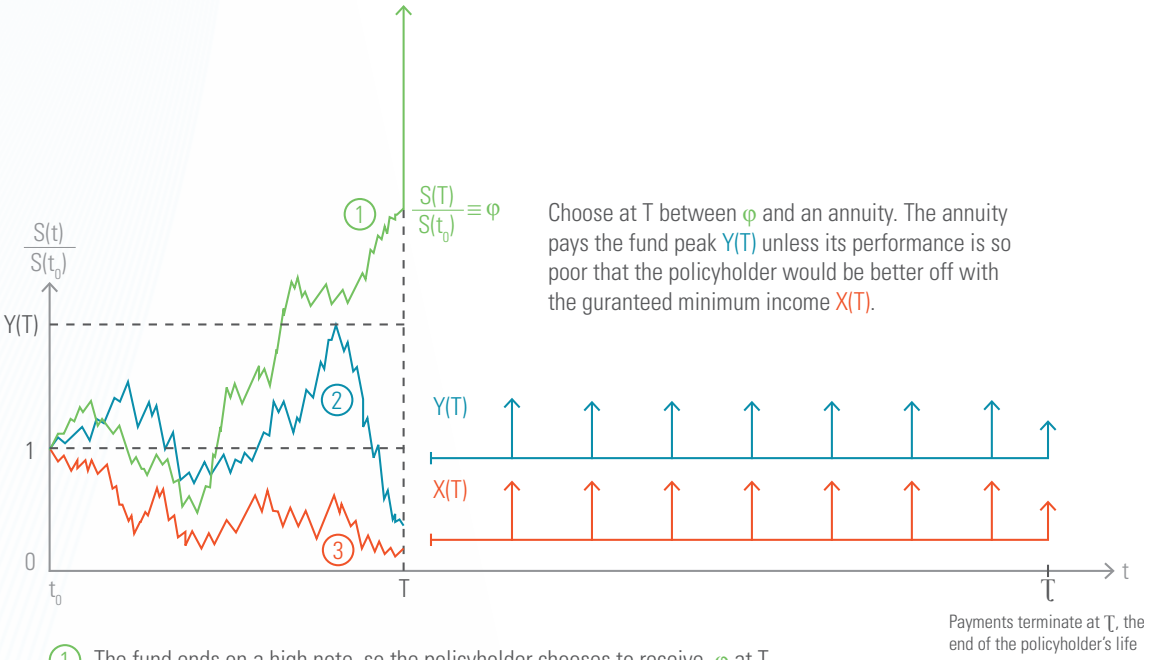

Fig. 3. Schematic illustration of a Guaranteed Minimum Income Benefit variable annuity policy

In our example, we choose the following numerical values:

Description	Value
Notional N	10000
Initial investment time t_0	July 31st 2012
Maturity time T	July 31st 2022, so $T - t_0$ is 10 years
Annual payment rate g	5%
Guaranteed rate g	5%

4.2.2 Product structure

We now structure the above trade using the *F3 Toolbox for use with MATLAB*. Our eventual goal is to construct a ***Product*** representing the choice between two other ***Products***; the single payout of *Eq. (3)* at and the annuity of *Eq. (4)*. But to construct those ***Products***, their payoffs must first be expressed using ***Indices***. The condition of *Eq. (4)* is a composite ***Index*** as described in *Subsec. 2.4*. The main underlying ***Index*** (see *Subsec. 2.3*) is an object that we decided to call *EquityFund* that encodes the valuation-relevant definition of the observable such as the currency in which it is denominated and the days on which its value can be observed. This object is displayed in *Fig. 4* using the *F3 Workstation*.

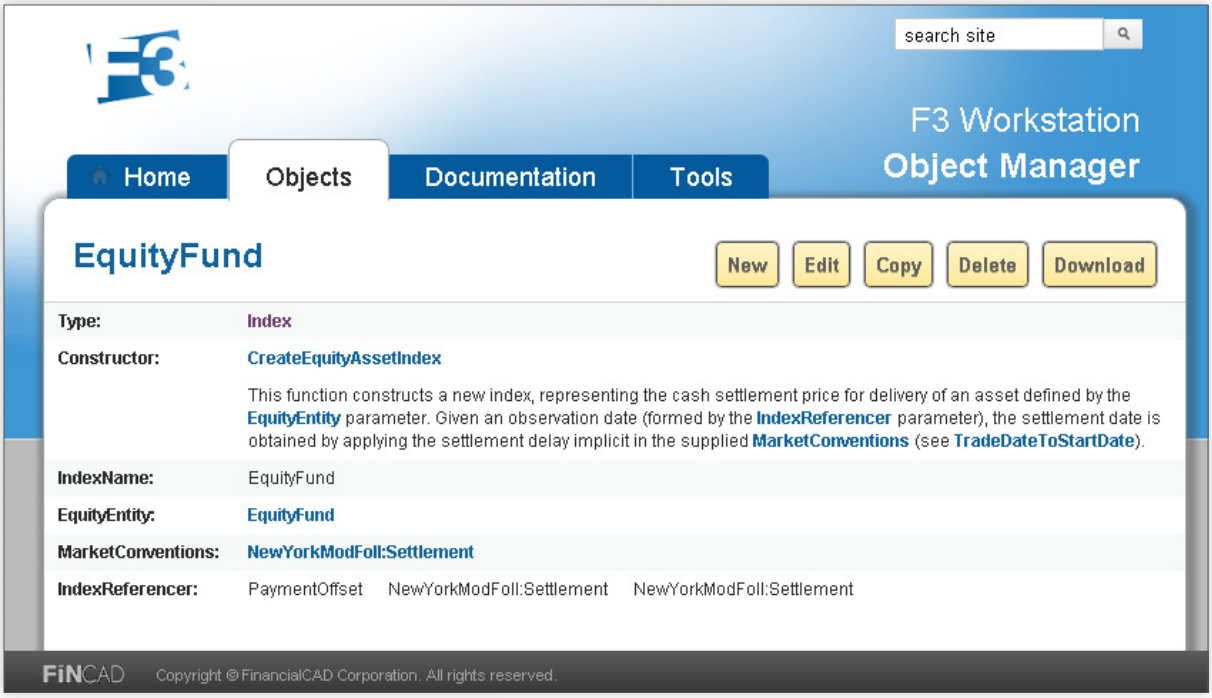

Fig. 4. F3 Workstation display of the Index object representing the equity fund underlying the example Guaranteed Minimum Income Benefit variable annuity policy

There is a second underlying that is relevant to the question of how much the GMIB policy is worth. We named it *PolicyholderSurvivalIndex* and it represents whether, at some future observation time, the policyholder is alive. It therefore takes values 0 or 1 at each observation time on each iteration of a simulation, and its expected value is the survival probability of the policyholder. Its F3 Workstation display is shown in *Fig. 5*

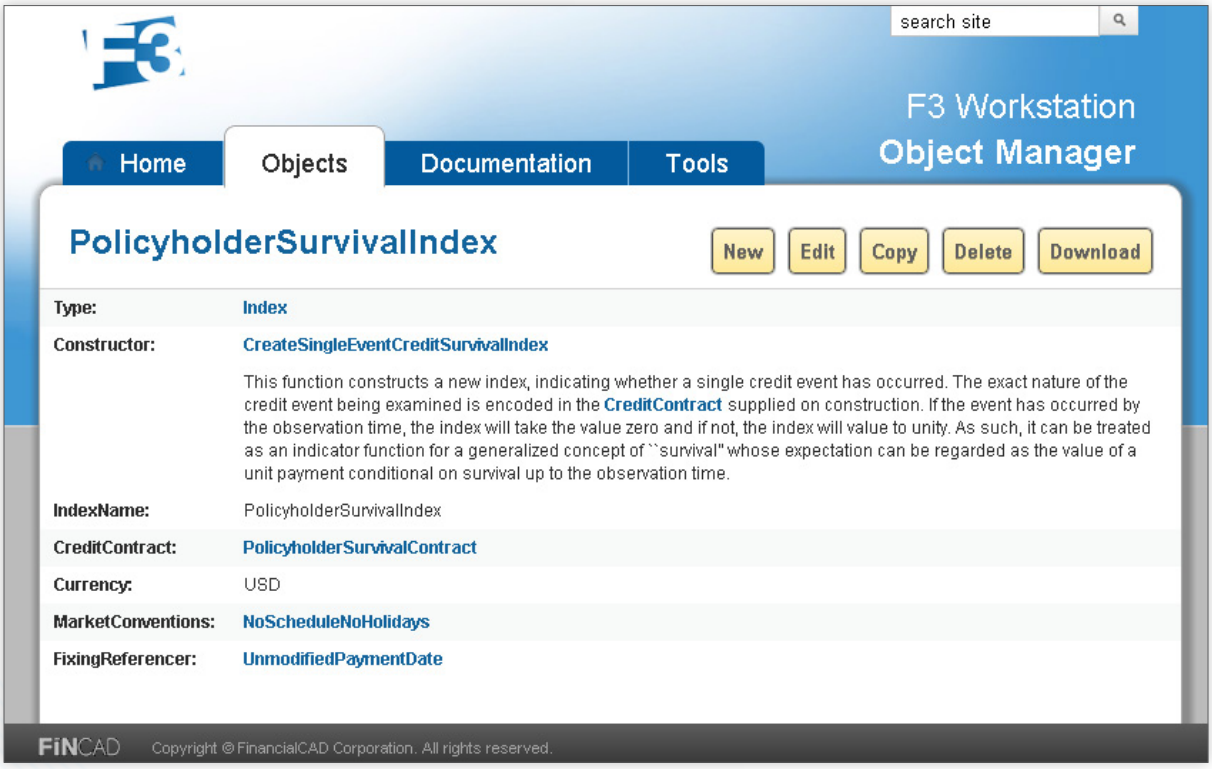

Fig. 5. F3 Workstation display of the Index object representing the survival of the policyholder

For brevity, and for consistency with the notation in the formulae in *Subsubsec. 4.2.1*, we will rename these objects:

```
F3.CreateDuplicateObject( 'S_t', ...      % new name
                          'Index', ...    % object type
                          'EquityFund' ); % old name
F3.CreateDuplicateObject( 'Q_t', ...
                          'Index', ...
                          'PolicyholderSurvivalIndex' );
```

The object `F3` above represents our view of the (multi-user) F3 library and allows us to call functions in the F3 API to construct, manipulate and use other objects. The subscript `t` in `S_t` and `Q_t` reminds us that these ***Indices*** have not yet been bound to a specific observation time. Although the name of the function suggests that a copy of each object is made, F3 does not waste memory. Only one underlying object exists, but it is now identified by two names. Although MATLAB does possess an object model, we are operating within the *named object* paradigm introduced in *Subsec. 5.2*. F3 objects are not bound to variables in MATLAB's memory, but are identified by means of names such as `S_t` and `Q_t`. For an example of a strongly-typed call to the *F3 SDK* in C++, see *Appendix A*.

The payoffs in *Eq. (3)* and *Eq. (4)* are functions of the normalized fund value $S(t)/S(t_0)$. To this end, it is useful to define a new ***Index*** in which the equity fund ***Index*** is bound explicitly to the observation point t_0

```
F3.CreateIndex( 'S_0', 'bind( S_t, 2012-07-31 )' );
```

A ***Product*** representing the obligation to make the payment in *Eq. (3)* can then be constructed as follows:

```
N = 10000.0;
T = F3.class.Date('31-Jul-2022');
F3.CreateSingleCashflowProduct( 'FundPayoff', ...      % Product object name
                                T, ...                 % payment date
                                'Q_t * S_t / S_0', ... % Index expression
                                N, ...                 % notional
                                'USD', ...             % payment currency
                                'Receive' );           % payment direction
```

Note that the ***Product*** part of the above statement stops at the level of stating that one cash flow will occur. It delegates the job of specifying the amount to an ***Index*** expression, which is the payoff of *Eq. (3)* made explicitly conditional on the survival of the policyholder. If the policyholder is not alive on July 31st 2022, then the payment amount is zero. The ***Index*** formed from the expression

```
'Q_t * S_t / S_0'
```

is observed ready for payment on July 31st 2022, which is equivalent to setting $t = T$ and results in *Eq. (3)*.

For the annuity payoff of *Eq. (4)*, we must express the guaranteed minimum income *Eq. (5)* and the fund's peak until time T, *Eq. (6)*. The guaranteed minimum income is a constant, which we can therefore pre-compute:

```
r_g = 0.05;
T_years = 10;
F3.CreateIndex( 'X_T', ( 1 + r_g )^T_years );
% precalculate this constant(about 1.63).
```

Constants fulfill the Index requirements and so we can construct `X_T` as a constant value. To find the peak value of the fund, we can do the following:

```
F3.CreateIndex( 'Y0', 1 );
t0 = F3.class.Date( '2012-07-31' );
for t = 1:T_years
   % calculate a valid date t years from t0
   d = F3.MaturityDate( t0, [num2str( t ) 'y'], 'NewYorkModFoll' );
   % numerator is, say, bind( S_t, 2015-07-31 )
   numerator = ['bind( S_t, ' datestr( d{1}.SerialDateNumber,
               'yyyy-mm-dd' ) ' )'];
   % fraction is, say, bind( S_t, 2015-07-31 ) / S_0
   F3.CreateIndex( 'fraction', [numerator ' / S_0'] );
   % Y3 is max( Y2, fraction )
   Yt = ['Y' num2str( t )]; Ytm1 = ['Y' num2str(t-1)];
   F3.CreateIndex( Yt, ['max( ' Ytm1 ', fraction )'] );
end
F3.CreateIndex( 'Y_T', Yt );
   % Y_T is the maximum value of S_t from time t0 to T
```

The above code loops through an annual schedule, constructing an index expression which can be illustrated schematically as

$$
\begin{aligned}
Y_0 &= 1 \\
Y_1 &= \max\left(\frac{S(t_1)}{S(0)}, Y_0\right) \\
Y_2 &= \max\left(\frac{S(t_2)}{S(0)}, Y_1\right) = \max\left(\frac{S(t_2)}{S(0)}, \max\left(\frac{S(t_1)}{S(0)}, 1\right)\right) \\
&\vdots \\
Y_T &= \max\left(\frac{S(T)}{S(0)}, Y_{T-1}\right)
\end{aligned}
$$

A version of the function accepting a collection of arguments as opposed to the binary function shown above would be an additional convenience, but is not necessary. In the case of `X_T`, the function `CreateIndex` was called with a constant argument. In the above code snippet however, we see it being called with Index expression arguments. In fact, throughout the entire F3 API, any argument whose type is ***Index*** accepts an expression.

We can now construct a *Product* representing the annuity itself; the obligation to make a series of payments according to a schedule.

```
T = F3.class.Date('31-Jul-2022');
schedule = F3.RollSchedule( T, '30y', 'ScheduleGenerator' );
g = 0.05;
F3.CreateSingleCurrencyFixedLeg( 'Annuity', ...       % Product object name
          schedule, ...              % payment schedule
          g, ...                     % fixed payment rate
          N, ...                     % notional
          'USD', ...                 % payment currency
          'Receive', ...             % payment direction
          'Q_t * max( X_T, Y_T )' ); % Index. T is fixed but t varies
```

The amount of a cash flow in this Product at time t is given by

$$N\alpha g Q(t)\max(X_T, Y_T)$$

where α is the accrual fraction associated with each payment, and contained within the schedule specification. In the present case of annual payments, α is close to unity. While we used some schedule-generation functionality within F3 to form the schedule, the Leg constructor consumed the schedule itself, decoupling the general problem of schedule generation from *Product* construction and enabling any custom schedule to be entered by hand if necessary. With the pair of underlying *Products* now constructed, we can form the variable annuity.

```
F3.CreateBinaryChoiceProduct( 'GMIB', ...       % Product object name
          'FundPayoff', ... % choose this Product
          'Annuity', ...    % or choose this Product
          T, ...            % choice time
          true, ...         % holder has right to choose
          'Receive' );       % holder chooses ownership of underlying
```

This *Product* represents the right to choose between the two underlying *Products* embedded within the Guaranteed Minimum Income Benefit term sheet. Having defined clearly the legal obligations and rights whose value we are to calculate, we can now proceed to the valuation.

4.2.3 Valuation

Fig. 2 shows that valuation in F3 requires three objects, a *Model*, *Product* and *ValSpec*. The focus of this paper is not modeling, so we will assume that a sufficient *Model* is provided. In *Subsubsec. 4.2.2* we showed the construction of the *Product* which means that only the *ValSpec* remains. We choose Monte Carlo simulation, configured as follows:

```
processes = { {'LiborUSD6m', 'HullWhiteOneFactor'}; ...
            {'S_t', 'Black'} }; % some simple modeling choices
F3.CreateSimulationValuationSpecification( 'HybridSimulation', ... % ValSpec
object name
          'Default', ...          % Underlying ValSpec
          'USD', ...              % numeraire is USD money-market account
          processes, ...          % modeling assumptions, see above
          'Sobol', ...            % underlying random number generator
          'AllCoresButOne', ...   % multithreaded calculation
          2^20 - 1 );             % iterations
```

In the above code we have constructed a *ValSpec* named `HybridSimulation` which will use default settings for all calculations except any involving U.S. dollar interest rates and the Index `S_t`. In particular, `Q_t` is absent from the list of processes for the simulation, which means that we choose to neglect any correlation between the life-expectancy of the policy holder and the value of the equity fund, taking policyholder survival probabilities from actuarial tables. Armed with this object, we can now value the policy:

```
value = F3.ValueProduct( 'Model', ...     % taken as read
          'GMIB', ...                     % the variable annuity Product
          'HybridSimulation', ...         % ValSpec constructed above
          {'Value','Variance','Histogram','RiskReport'} ); % requests for
          output
% pretty-print the value
disp( [ value{1,1}, ' ', num2str( value{1,2}, '%.2f' ) ] );
USD 11192.87
```

In the list of requests, in addition to the Value of the policy and its Variance, we have requested a Histogram and a RiskReport. The histogram is shown below in *Fig. 6* and the risk report is discussed in *Subsubsec. 4.2.4*.

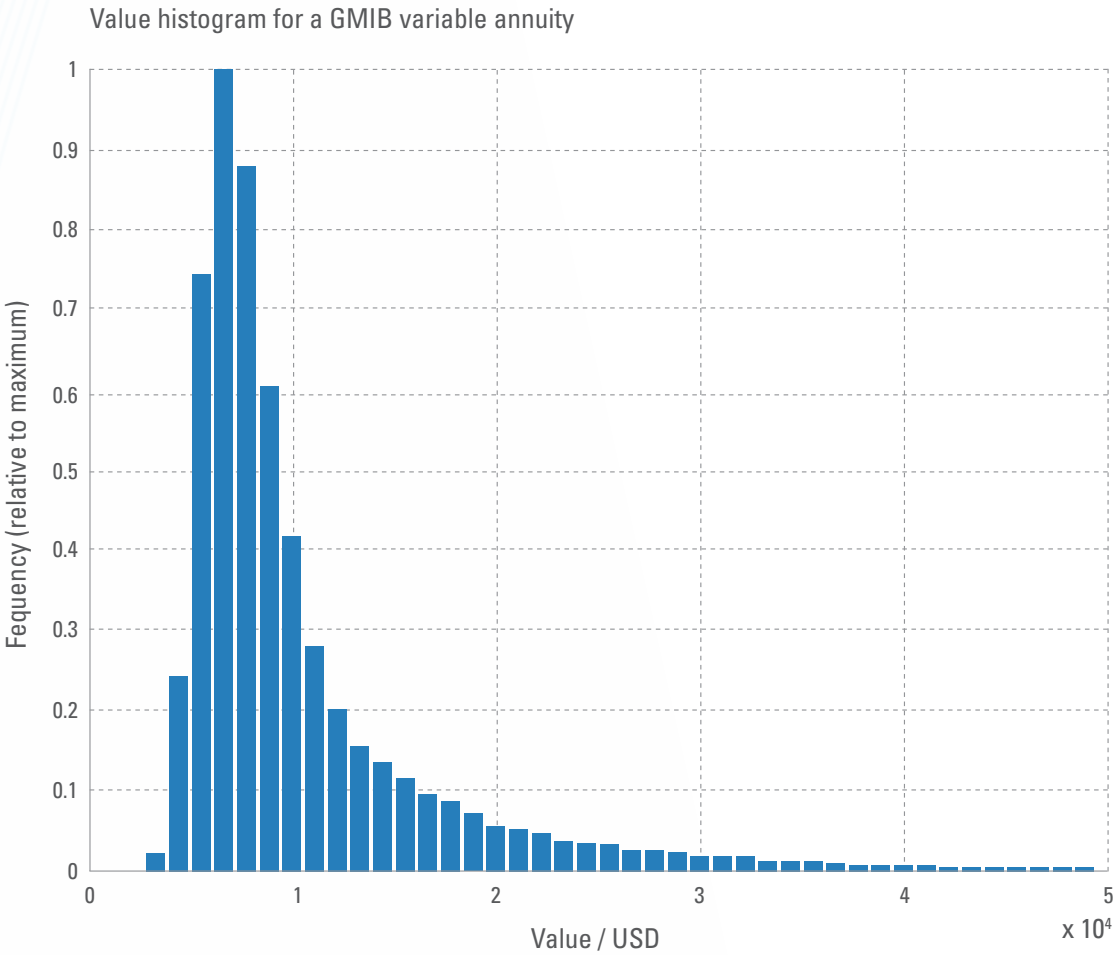

Fig. 6. Histogram of value of the variable annuity

4.2.4 Exposure

One of the requests supplied to the valuation call above is RiskReport. If V is the value of the variable annuity *Product* and the set of M market quotes on which this value depends is denoted $\{\alpha_i\}\ i = 1 \ldots M$, then the risk report gives the first-order (linear) term in the relationship between V and $\{\alpha_i\}$

$$(7) \qquad \left\{\frac{\partial V}{\partial \alpha_i}\right\} i = 1 \ldots M.$$

While it is possible to approximate these derivatives with finite differences, it is prohibitively expensive in general, amounting to M times the cost of an individual calculation of V. In the current example, $M = 106$, but for a cross-asset portfolio M can easily exceed 1000. By contrast, requesting the risk report for the GMIB increases the computational cost by less than one third.

The exposure to each quote in the risk report takes the following form:

MarketDataName	MarketDataType	QuoteSpecification	Quote	Currency	Exposure	ExposureType	InstrumentType
USD	SwapRates	5y	1.081%	USD	793.12	<RawValueExposure>	InterestRateSwap
USD	SwapRates	5y	1.081%	USD	0.0793	PVBP Delta	InterestRateSwap
USD	SwapRates	5y	1.081%	USD	160.26	EquivalentNotional	InterestRateSwap

The first four columns describe the quote to which exposure has been computed, in this case the 5 year swap rate of 1.081%. They are the same for each of the three lines in the above table because the whole table concerns the exposure to the same rate. The first row, whose exposure type is <RawValueExposure>, gives the raw partial derivative of *Eq. (7)*.

In the second row, it is scaled by 1bp in order to give, to first order, the finite change in V resulting from a 1bp change in the rate

$$(8) \qquad \Delta V = \frac{\partial V}{\partial \alpha_i}\Delta\alpha_i + O\left((\Delta\alpha_i)^2\right).$$

For small bump sizes like 1bp, the second order term is usually negligible. Indeed, a bump size of 1bp is usually chosen for finite difference calculations in order to approximate the exact local exposure given by the linear term in *Eq. (8)*.

The third row gives the amount of notional required for the quoted trade, in this case the 5 year swap, to hedge away the exposure to changes in its quote. If the value of a 5 year vanilla interest rate swap paying coupon c with notional N is $I(N,c)$, then this row of the risk report gives W such that:

$$\frac{\partial\left(V + I(W,s)\right)}{\partial s} = 0$$

where s is the quoted swap rate, $I(N,c) = 0$.

Given the high fidelity afforded by the risk report in F3, we can survey the entire exposure of the GMIB, rather than guess where we should dedicate valuable computation time. For example, we see that for each percentage point change in the correlation between the Brownian motions driving the rates and equity processes, V only increases by about $21; a relatively flat exposure.

In *Fig. 7* we can see that the largest exposures to the vanilla rates instruments (cash deposits up to 9 months, futures, then swaps) used to build the discount curve, fall on and after 10 years.

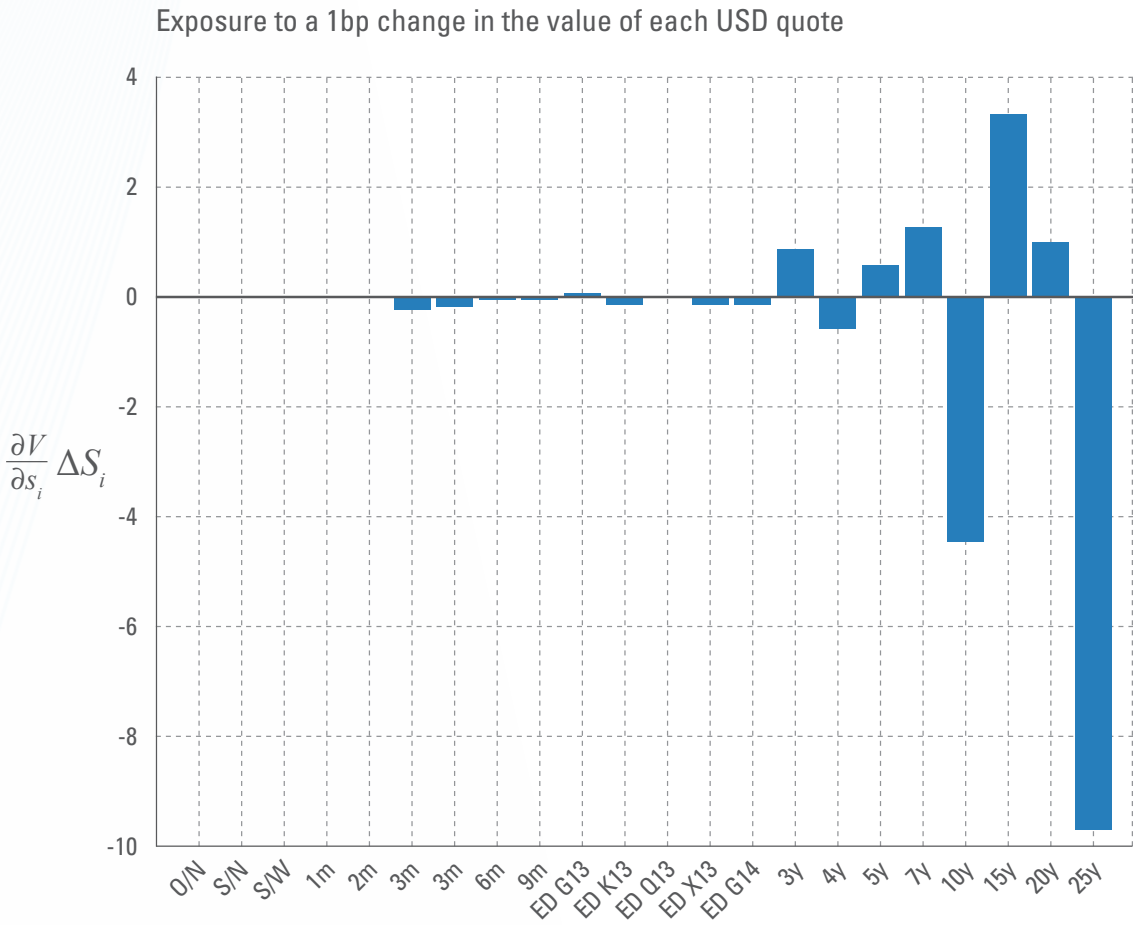

Fig. 7. Exposure of the GMIB to discount instrument quotes

Similarly, we can examine the exposure to the policyholder survival probabilities that underlie the valuation. *Fig. 8* shows relatively modest exposure overall, with a profile that reveals that remaining alive long enough to make the choice has a roughly equivalent affect on V to collecting the annuity over the following thirty years.

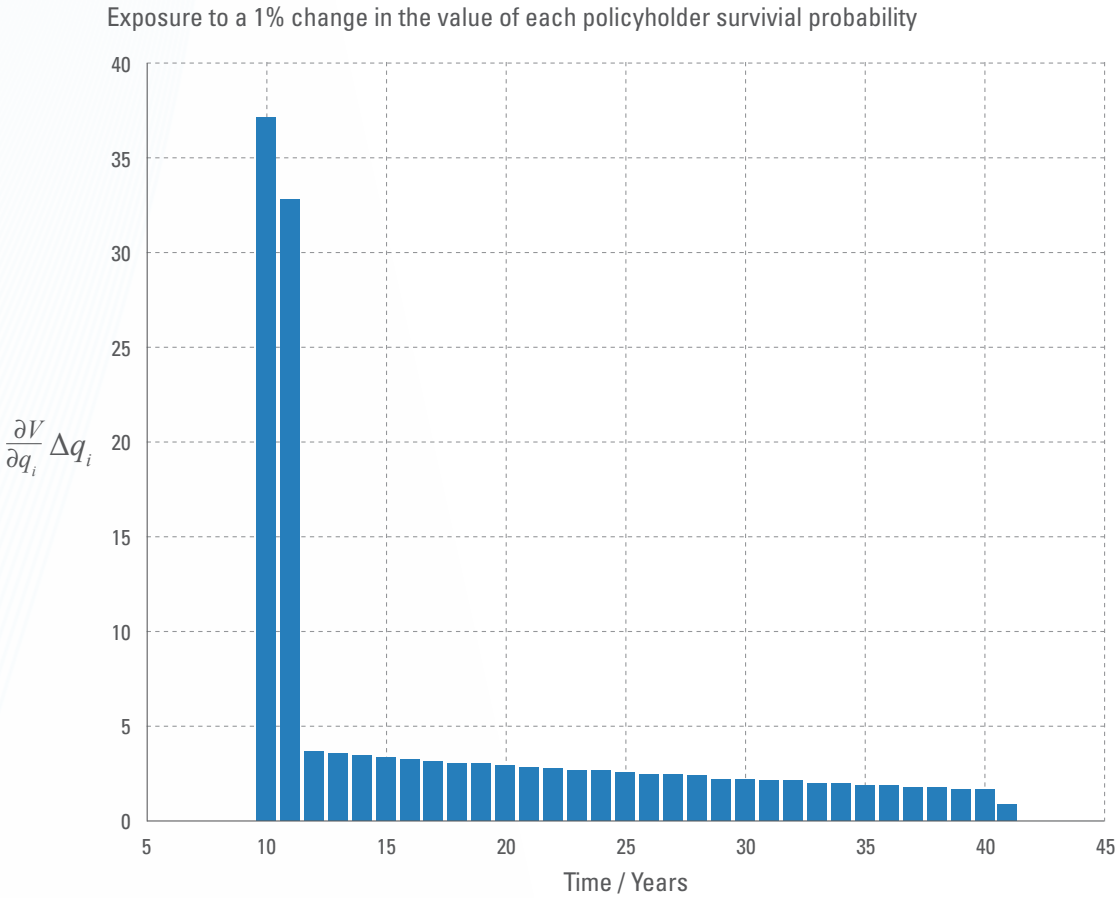

Fig. 8. Exposure of the GMIB to policyholder survival probabilities

5 TOOLS AND TECHNIQUES

The focus of this paper is the conceptual structure of financial derivatives' valuation and how that structure must inform the design of any modern analytics platform. In addition to addressing issues specific to this problem domain, however, we cover some considerations of tools, process and architecture which are more general, applying to software systems in this and other domains.

5.1 Programming paradigm

The first consideration is the choice of programming paradigm. *Object-oriented* programming has enjoyed ubiquitous success for solving software engineering problems for the last thirty years and continues to do so. It has an undeserved reputation of being complex and hard to understand. An object is simply an organized unit of data in a program that represents something in real life as closely as possible. For example, a circle object in a geometry program might have the capability to calculate and report its area, radius, diameter and circumference. It is the ability to define new types, or classes of object, so as to represent closely real-life phenomena, that gives object-oriented programming much of its power.

It is when constructs in a program mirror the essential aspects of the problem at hand, that the program solves it most effectively. In this paper we have defined concepts (or abstractions) such as *Product*, *Index* and *Model* which underpin the domain of derivatives valuation. In an object-oriented program, each concept can be represented in code as a class of object, able to perform relevant operations and interact with other objects in the same way that those concepts relate to each other in the real world.

A practical concern is the choice of computer language for the implementation of a system. A vast array of languages support object-oriented programming. In any numerical computation one of the primary requirements of an implementation language is *high performance*; the ability to construct a system which is able to make the most of the available hardware resources. The C++ language [8] satisfies this requirement and has been the defacto standard for production systems in quantitative finance for many years. So much so, that while high performance is the fundamental requirement, the use of C++ itself has become a proxy for that requirement in much of the financial domain.

5.2 Engineering challenges and their solutions

Modern C++ is a formidable tool that puts great power in the hands of software developers and can yield efficient and elegant solutions to programming problems. However, there are a number of pragmatic issues which must be addressed, some specific to systems implemented in C++, others more general. We now address each one in turn.

- **ABI compatibility**
  If a library is written in C++, then client code (which calls the library) must be compiled with the same compiler tool-chain as the library itself. The artifacts produced by a compiler that are relevant to calling a library from client code are known as the Application Binary Interface (ABI) and this issue is known as the ABI compatibility problem in C++. While this may not represent such an onerous constraint for an organization's internal systems, it is critical for the software vendor to solve this problem, particularly for a multi-platform library.

  A simple solution is to keep C++ as an implementation detail; hide the fact that it was used from any calling code, by exposing a C, not C++, interface. The C language

is planet Earth's "lowest common denominator", the de facto standard for systems' implementation. By choosing C for the interface it may seem that we are forfeiting the use of objects, but that is not the case as we shall soon see.

- **Technical support**
  Another practical matter that is more acute for the software vendor, but nevertheless present for the group providing a system for internal use within an organization, is the provision of technical support. In order to diagnose problems it is necessary to reproduce them reliably. If a system runs on multiple platforms and can be called from multiple language environments, then it can become costly to manage such reproductions.

  An efficient solution to this reproduction problem which also eliminates the impact of platform heterogeny would be provided by a mechanism with (a) the ability to record both the information flowing into the library and the information content of the results returned by it and (b) the ability for the library to consume such records and execute the same series of calls. Further practical advantages would be obtained with a recording format that was human-readable, in addition to being merely "machine-replayable".

- **Regression testing, transparency and auditing**
  Upgrading software to a newer version always carries risk. Bug-fixes may result in different numbers being produced by a calculation. Library users must be aware of any such changes. In addition, particularly within the financial sphere, it is often necessary to report the details of computations to satisfy regulatory bodies, auditors or stakeholders

  The recording (and replaying) mechanism described above provides an elegant solution to these problems also. It becomes possible to record an audit trail of every piece of information that flows across the library's interface in either direction. Regression testing is facilitated by comparing the results of executing the same set of function calls in each version of the library.

  These use cases place two additional requirements on the recording mechanism. Firstly, it must not be possible for only a subset of the functions that form the library's interface to be supported due to human error. In matters of regulation and auditing, there must be a guarantee of 100% coverage. Secondly, the recording format should be easy to parse, to ease the construction of regression testing tools. Choosing an established format such as XML would allow the use of off-the-shelf tools for reading and manipulating recorded function calls and results.

- **Distributed computing, persistence, sharing**
  It has been several years since processors switched from faster clock speeds to using multiple processors to increase computing power. Any modern high-performance system must address the problem of distributing its calculations over multiple cores within a machine and across multiple machines across a network. The transport of simple data types (numbers, character strings, Booleans etc) is trivial. But when objects are used to represent real life (and therefore often complex) concepts, explicit work is necessary to facilitate network transport.

  The technical term for converting an object in memory to a form compatible with network transport is *serialization*. In fact, the problem of communicating an object across a network is equivalent to that of saving an object to non-volatile memory such as a hard disc. Such persistence forms a critical part of the robustness of any server and is even at the heart of the everyday act of a user saving her work to a file.

Many robust serialization frameworks are available in C++. However, serialization is a distraction, a necessary but costly evil, not something which provides direct value to solving the business problem in a given domain, such as the pricing of financial derivatives. In that domain, it is rare to find a quant who, in addition to expertise in financial mathematics, is well versed in object serialization.

This problem is yet another which can be resolved neatly with the recording mechanism already described, by making the serialized form of an object consist of the function call that is necessary to construct it. Other objects may have been supplied as input arguments to the object's construction call and so some straightforward dependency tracking is necessary. By choosing this format, the need for quants to have any knowledge of serialization is eliminated entirely.

Strictly, an object's internal state is only equivalent to its constructed state if it is immutable. Immutable objects also vastly simplify the engineering of concurrent systems which support multiple users, threads and processes. As a final technical note, bitwise immutability is not strictly necessary, which means that some bespoke serialization may be desirable, but this is strictly an optimization.

- **Deployment and application development**
  An established pattern followed by well-engineered applications, particularly those with Graphical User Interfaces (GUIs), is to draw together a collection of subsystems (libraries) which contain the logic that solves the given business problem. The construction of GUIs is made more efficient if a library can describe itself, in order to avoid the proliferation of hard-coded artifacts where the results of runtime queries of the underlying libraries would suffice instead.

  The common term for self-description in software libraries is reflection. Reflection is greatly facilitated by specifying library properties (such as its API) as metadata and then generating code from it, rather than having developers author such code outright. The aggressive use of metadata and code generation also eliminates the need for manual labor to support the deployment of a library in multiple calling environments, allowing a single codebase to be called from, for example, .NET, Java, C++, C, MATLAB and Microsoft Excel without any per-environment cost to extending the library.
- **Object-oriented API without an object model**
  Some of the above calling environments, such as Excel, do not possess an object model and yet we have been discussing the merits of allowing a user to manage complexity by creating and manipulating internal state within a library being called from such an environment. The most common solution to this problem is to simply provide some form of handle mechanism. For example, we might require that when a new object is constructed, the user supplies a character string which is used to access the object and instruct other parts of the library to use it. In other words, the user would give objects names.

  While this solves the object management problem in environments devoid of an object model, it may seem unnatural to name objects in environments which themselves support object-oriented programming. A solution here is to regard named objects as the lowest-level interface, and generate (using the library's reflection) the code for higher-level interfaces, native to a given environment. Such code would manage object names, while providing a strongly-typed interface to client code.

  A strongly-typed interface requires that each input argument has a well-defined type,

which imposes a further challenge on the underlying named-object API, which must cope with calling environments devoid of any type system. A solution to this problem is to regard each type as a specific collection of simply-typed data such as numbers and strings (where the names of objects can be included in the set of strings) plus a rule for constructing a C++ object from the data. Multiple rules, consuming different collections of simple data, can construct the same type of object, each one corresponding to a different constructor. To hold the data itself, some form of array of variants is most appropriate. MATLAB's mxArray or Excel's XLOPER are examples of variant arrays.

- **API extensibility**
  Different calling environments impose different constraints on the extensibility of an API. The core API of a library servicing all such environments should be able to accommodate them all. One example would be that optional arguments in some languages must follow mandatory ones. Another would be the absence of any notion of overloading in some languages.

  One of the most flexible data structures that we can utilize to encode the arguments to a function call is a collection of name-value pairs, particularly when the value is an array of variants. Such a collection does not require an ordering, although the pairs can be ordered if desired.

Appendix A gives an example of a functional API that embodies the ideas covered in this section. The primary focus of this paper, however, is not the aspects of design that are common to many domains, but the specifics of analytics platforms. The topics covered in this section can be regarded as the "plumbing" needed to deliver solutions to a business problem (that of derivative valuation) and are of particular interest when considering the problem of integration into an existing enterprise infrastructure [9].

6 CONCLUSION

The pricing and calculation of exposure of derivatives is a complex computational problem whose relevance has heightened since the credit crisis of 2008. The need to account for the effect of counterparty exposure means that sophisticated pricing techniques once applied only to exotics are now necessary to manage vanilla portfolios. This problem has a solution: analytics technology with the flexibility and sophistication to cope with such complexity.

The traditional approach of quants writing code to price specific trades or classes of trade has become prohibitively costly because too much time and effort is required to accommodate changes in trade structure. Highly paid quants are effectively asked to do repetitive manual labor - writing bespoke code to value a new trade type or to account for a new risk factor. The task of designing for modularity and reuse is often left to the quants, among whom it is rare to find one with the necessary broad architectural vision in addition to the already rare skills of numerical computing and mathematical modeling.

In this paper we have considered precisely this problem of analytics platform architecture; the design considerations, constraints and conceptual structure that must form the foundation of any analytics platform with the flexibility to price a generic derivative. Our focus was trade description, not modeling, although we summarized the basic requirements of a generic valuation capability.

Finally, we gave the example of F3, a concrete embodiment of the best practice presented in this paper, and applied it to the problem of pricing a Guaranteed Minimum Income Benefit variable annuity policy, as an arbitrary example. We described the policy using the concepts of *Product* and *Index*, specified the valuation approach in a *ValSpec* and, because modeling was not the focus of this paper, used a pre-built *Model*. In addition to the policy's value, we calculated its exposure more than 100 times more quickly than in conventional approaches.

The flexible and generic nature of F3 yields a system that meets the steep demands that rapid innovation and post-crisis regulatory scrutiny place on the quantitative analysis of financial derivatives in today's markets and tomorrow's.

APPENDIX A. F3 API STRUCTURE

Subsec. 5.2 tackles many of the pragmatic issues faced by object-oriented libraries across multiple problem domains such as ABI compatibility, support, regression testing, transparency, auditing, distribution, persistence, API extensibility and using the library where an object model does not exist. These problems are solved in F3 by choosing an XML-based API as the most fundamental form of information transfer in and out of the library. With a sufficiently rich reflection capability, higher-level APIs can then be generated automatically. In this section we examine the lowest and highest level APIs in turn.

Sec. 4 describes a call to `ValueProduct` made from Excel. A similar call, but made using the lowest-level XML-based API takes the following form

```
fincad::interface::Library Lib( "f3cpp:" ); // product licensing information
fincad::interface::Context F3( Lib, "2.8" ); // versioning information

string_t call = "<f>" // function call
          "<n>ValueProduct</n>" // whose name is ValueProduct
          "<a>" // and in whose argument list
             "<p>" // the first parameter
                "<n>Model</n>" // has name Model
                "<v>" // and (2d variant array) value whose
                   "<r><s>TodaysModel</s></r>" // first and only row
                   //contains the string TodaysModel
                "</v>"
             "</p>"
             // and similarly for subsequent arguments
             "<p><n>Product</n><v><r><s>VanillaSwap</s></r></v></p>"
             "<p><n>ValuationMethod</n><v><r><s>Default</s></r></v></p>"
             "<p><n>Requests</n><v><r><s>Value</s></r></v></p>"
          "</a>"
          "</f>";

std::pair< string_t, bool > result = F3.callFunction( call.c_str() );
```

If the Boolean in the return value is false, then the call was not successful and the result contains an error message. Otherwise, the result is

```
"<r><s>USD</s><d>41530.25</d></r>"
```

For a full description of the grammar and syntax of the above XML-based encoding, called F3ML, see [10]. Because all calls go through the single entry point of the `callFunction` member function of `F3`, we can guarantee that all calls and results can be logged. The implementation of a tool that replays such logs is trivial; it just has to read a line from the log and execute it. The choice of XML for encoding means that the format is also human-readable and there is a wide array of established tools for manipulating call and result logs. The serialization problem is also solved if objects cache the call with which they are constructed, together with the names of objects on which their construction depends.

What about performance? Surely the conversion of data, particularly numerical data to a string representation and back on every call imposes an unacceptable overhead for a high-performance numerical computation library? This is a common gut-feel reaction but turns out to be a premature optimization. Such overhead (measured in microseconds for most calls) is

negligible, because only a small amount of numerical data passes through the API relative to the typical cost of computation. Not only that, but numerical data (such as market quotes) tends to be entered once and then persisted in memory (say, within a market data object) and it is the name of this object which is passed in function calls. Most objects, large and small, are constructed once and then manipulated by passing references too them. Further performance gains coming from the efficiency of *Models* (*Subsubsec. 3.1.1*), the ability to calculate *full analytic exposure* and multithreading mean that the XML-based structure above is the clear optimum in the range of possible design choices when considering the total cost of addressing the issues described in *Subsec. 5.2*.

Encoding every function call as XML, however, is not necessarily how the end-user wishes to operate. For that reason other higher-level APIs can be generated which encode and decode calls and results as XML. One level above is to operate with some form of variant array such as in Excel, as shown in *Fig. 2*. The MATLAB code examples of *Subsec. 4.2* also work with variants. Variant data allows for the kind of flexibility that is required in a modern analytics platform, but has the disadvantage that it is weakly-typed, so that error messages resulting from invalid input only appear at runtime. One of the strengths of strongly-typed languages such as C++ is that the compiler enforces correctness when the code is compiled, before it is run. Runtime errors can still exist, but type errors can be caught earlier.

The following code sketches an example of F3's strongly-typed C++ API.

```
// take model and valspec as read
fincad::f3::index
payoff ("max(0, min(min(R_A,10%)+min(R_B,10%)+min(R_B,10%), 20%))");
Product product = Product::CreateSingleCashflowProduct( F3,// Context object
          Object::Date("2012-08-23"), // payment date
          payoff,
          1.0e6,                      // notional
          "USD",                      // currency
          fincad::f3::payrec::Buy() ); // direction

typedef valuation_engine_requests requests_t;
requests_t requests = requests_t::Value() + requests_t::RiskReport();

ProductValue result = model.ValueProduct( product, valspec, requests );
std::cout << result.Value() << std::endl << result.RiskReport();
```

The most important feature of this code is that no name is chosen by the user for the single cash flow product object when it is constructed. Instead, we have a C++ type `Product`. The Context object `F3` is supplied on construction because it acts as a scope - it tracks every object in F3, facilitating the use of tools such as the *F3 Workstation*, as shown in *Fig. 4*, even in user applications built using the *F3 SDK*. The `product` object above is very lightweight, effectively acting as a proxy for the name.

The object `payoff` is not a string, although one of its constructors does consume a string (see *Eq. (1)*). The objects `product` and `result`, likewise, are dedicated C++ types with member functions, type checking and overloads as with any other C++ type. The above code shows that the act of valuation has been associated with the *Model* type and is no longer a free function.

This expressive code achieves a concise and high-level expression of the artifacts necessary for analytics computations while retaining the flexibility of variant-based input, but without paying the price of weak typing.

BIBLIOGRAPHY

[1] Black, F. and Scholes, M. (1973), The Pricing of Options and Corporate Liabilities, Journal of Political Economy, 81, 637-659

[2] Harrison, J. M. and Pliska, R. (1981), Martingales and Stochastic integrals in the theory of continuous trading, Stochastic Processes and their Applications, 11(3), 215-260

[3] Curves, FINCAD (2012, forthcoming)

[4] Generic Monte Carlo Simulation in F3, FINCAD (2012, forthcoming)

[5] Universal Risk in F3, FINCAD (2012, forthcoming)

[6] Marshall, C. et al. (2010), Valuation of a Guaranteed Minimum Income Benefit, North American Actuarial Journal, 14, 38-58

[7] Scenarios and Portfolio Risk Analysis in F3, FINCAD (2012, forthcoming)

[8] The C++ Standard, http://www.open-std.org/jtc1/sc22/wg21

[9] F3 Reference Architecture, FINCAD (2012, forthcoming)

[10] The F3 Reference Manual, FINCAD

ABOUT FINCAD

Founded in 1990, FINCAD provides advanced modelling solutions built on award-winning, patent pending technology. With more than 4,000 clients in over 80 countries around the world, FINCAD is the leading provider of financial analytics technology, enabling global market participants to make informed hedging and investment decisions. FINCAD provides software and services supporting the valuation, reporting and risk management of derivatives and fixed income portfolios to banks, corporate treasuries, hedge funds, asset management firms, audit firms, and governments. FINCAD Analytics can be accessed through Excel, MATLAB, as a Software-as-a-Service or embedded into an existing system through software development kits. Now, over 70 FINCAD Alliance Partners embed FINCAD Analytics within their solutions. FINCAD provides sales and client services from Dublin, Ireland, and Vancouver, Canada.

ABOUT THE AUTHORS

Dr. Mark Gibbs is the Chief Software Architect at FINCAD, and a key driver of FINCAD's award winning F3 solution. Dr. Gibbs has nearly 20 years of financial industry experience, including various roles in leading investment banks in London. Prior to joining FINCAD, he led the structured credit quantitative team at UBS. He holds a PhD in Theoretical Physics from the University of Cambridge and has published nearly 50 academic papers. Dr. Gibbs has served as an industry advisor on a number of projects, most recently for an EU research project on Monte Carlo methods.

Dr. Russell Goyder is the Director of Quantitative Research and Development at FINCAD. Before joining FINCAD's quant team in 2006, he worked as a consultant at The MathWorks, solving a wide range of problems in various industries, particularly in the financial industry. In his current role, Dr. Goyder manages FINCAD's quant team and oversees the delivery of analytics functionality in FINCAD's products, from initial research to the deployment of production code. Dr. Goyder holds a PhD in Physics from the University of Cambridge.

CONTACT:

Corporate Headquarters
Central City, Suite 1750
13450 102nd Avenue
Surrey, BC V3T 5X3
Canada

EMEA Sales & Client Service Center
Block 4, Blackrock Business Park
Carysfort Avenue, Blackrock
Co Dublin, Ireland

USA/Canada 1.800.304.0702
Europe 00.800.304.07020
London +44.20.7495.3334
Dublin +353.1.400.3100
Elsewhere +1.604.957.1200
Fax +1.604.957.1201
Email info@fincad.com
www.fincad.com

FINCAD®

FinancialCAD

TXA12W073-1