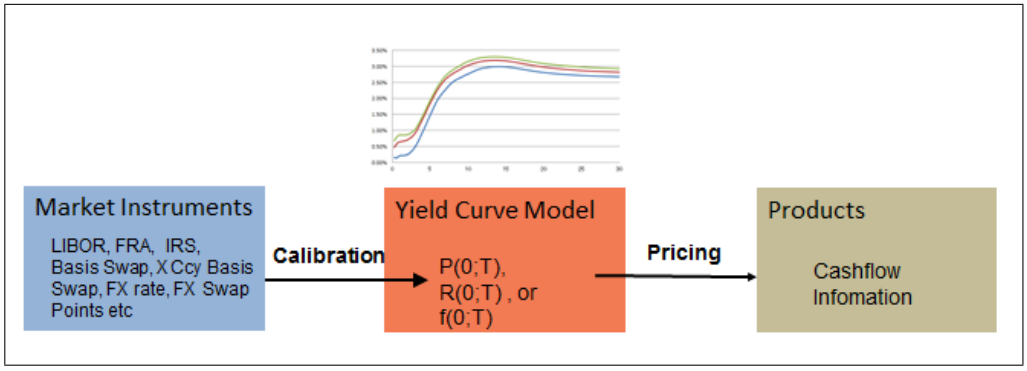
C++ によるデリバティブ・ライブラリ構築の基礎
6. イールドカーブ

高田勝己*

2018 年 12 月 25 日

1　イールドカーブの構築

マーケットでトレードされている流動性のある商品の価値をイールドカーブでプライシングを行い、その価値がマーケットでトレードされている価値になるように、イールドカーブの情報を逆算する。



(Figure 1) イールドカーブモデル、プライシング、及びカリブレーション

2　キャッシュフローのプライシング

$D(T)$ を現在 (時点 0) における時点 T のディスカウント・ファクターをする。

2.1　ビュレット (Bullet)・キャッシュフロー

時点 T における金額を A とすれば、現在価値 V は

$$V = D(T) * A$$

である。

* 株式会社 Diva Analytics, ktakada@divainvest.jp

2.2 固定金利のキャッシュフロー

元本 N、年率表示の金利を r、利息期間のスタート時刻を年で T_s、利息期間のエンド時刻を年で T_e とする。

$$V = N \cdot r \cdot (T_e - T_s) \cdot D(T_e)$$

2.3 変動金利のキャッシュフロー

現在まだ決まっていない、利息期間 $[T_s, T_e]$ の変動金利の現在価値を考える。この変動金利は時点 T_s で決まり、時点 T_e に支払われる。利息期間のスタート時点に N 単位のキャッシュをこの変動金利で運用すると、フィクシングした金利を L とすれば、エンド時点の T_e では $1 + L(T_e - T_s)$ となっている。これらのキャッシュフローは、

$$\begin{cases} \text{時点 } T_s: & -N \\ \text{時点 } T_e: & N \cdot (1 + L * (T_e - T_s)) \end{cases} \tag{1}$$

である。これらのキャッシュフローの総現在価値はゼロであるから、変動金利の現在価値を V をすれば、

$$0 = N\left(D(T_e) - D(T_s)\right) + V$$

これより、まだ決まっていない変動金利キャッシュフローの現在価値 V は

$$V = N\left(D(T_s) - D(T_e)\right) \tag{2}$$

と計算できる。

利息期間が $[T_s, T_e]$ である現在の変動金利のフォワードレート F を算出しておこう。F はフォワードレートの定義から

$$V = N \cdot K \cdot (T_e - T_s) \cdot D(T_e)$$

が成り立つような K であるから、(2) より

$$F = \frac{1}{T_e - T_s}\left(\frac{D(T_s)}{D(T_e)} - 1\right) \tag{3}$$

よって、(3) から、フォワードレート F を計算して、これがあたかも固定レートのように現在価値を計算しても同じことである。つまり、

$$V = N \cdot F \cdot (T_e - T_s) \cdot D(T_e) \tag{4}$$

3 Forward declaration （前方宣言）

関数を定義とは別に宣言できたのと同様に、クラスも定義なしに宣言できる。この宣言を **Forward declaration**（前方宣言）という。

```
class Matirx;
```

は Matrix という名前はクラス型であることをプログラムに教えている。この Matrix 型は、定義がまだ明らかではないので、不完全型 (**incomplete type**) という。不完全型は次の 2 つの場合に使われる。

- ある型に対するポインタや参照を定義するときは、その型は不完全であってもよい。
- ある関数を (定義ではなく）宣言するときに用いるパラメータの型や戻り値の型は不完全型でよい。

次の場合は、クラスは定義されていなくてはならない。

- ある型のオブジェクトをつかうコードを書くときや、ポインタや参照からクラスのメンバにアクセスするとき
- クラス定義で、データメンバの型（クラス）を宣言するとき

よって、あるクラス定義で、同じクラス型のデータメンバを持つことはできない。しかし、同じクラスのポインタや参照はメンバーにすることができる。

4 Cast

浮動小数点数の割り算をおこないたいときがある。

```
int i, j;
double slope = i / j;
```

この場合、**Cast** で陽な型変換をおこなう。

4.1 Old-style cast

```
double slope = (double) (i) / j;
```

4.2 Static cast

```
double slpoe = static_cast<double> (i) / j;
```

4.3 Dynamic cast

4.3.1 ポインタ型の Dynamic cast

dynamic_cast<T *>(e)、ここで T はあるクラス型の派生クラスで e はベースクラスを指すポインタである。

```
if (Derived *dp = dynamic_cast<Derived *> (bp))
{
    // dp が指す Derived オブジェクトを使う
}
else
{
    // bp が指すオブジェクトを使う
}
```

もし上で、bp が Derived のオブジェクトを指しているのであれば、キャストで dp は bp が指している Derived オブジェクトを指すように初期化される。そうでなければ、キャストの結果は 0(null pointer) となる。

4.3.2 参照型の Dynamic cast

dynamic_cast<T &>(e)、ここで T はあるクラス型の派生クラスで e はベースクラスを指すポインタである。

参照型の Dynamic cast がポインタ型の Dynamic cast と違うところは、エラー処理である。もし、ダイナミックキャストが失敗したら、キャストは std::bad_cast という例外を投げる。

5 コード

5.1 UTModelBase.h

```
#ifndef UT_MODEL_BASE_H
#define UT_MODEL_BASE_H
#include <string>
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
// UTModelBase
//
// The base class for all models.
//
class UTModelBase
{
public:
    static std::string const ourClassTag;

    // Friend class.
    // Make UTSolveForModelComponent a friend so that model can be calibrated.
    friend class UTModelFactory;

   friend class UTSolveForModelComponent;

    // Destructor.
    virtual ~UTModelBase() {}
    // Constructor.
    UTModelBase() {}

    // Clone.
    virtual UTModelBase* clone() const = 0;

    // Functions.
    virtual std::string classTag() const = 0;
    virtual double df(double time) const = 0;
    virtual double forwardRate(double startTime, double endTime) const = 0;

    // Return a reference to the ith sub-model
```

```
    virtual const UTModelBase* subModel(unsigned long i) const;

private:
    // solver in the calibration only use this.
    virtual void setComponent(unsigned int i, double component) = 0;
};
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
#endif // UT_MODEL_BASE_H
```

5.2 UTModelBase.cpp

```
#include "UTModelBase.h"
using namespace std;

const string UTModelBase::ourClassTag = "Model Base";

////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
const UTModelBase* UTModelBase::subModel(unsigned long i) const
{
    return nullptr;
}
```

5.3 UTModelYieldCurve.h

```
ifndef UT_MODEL_YIELD_CURVE_H
#define UT_MODEL_YIELD_CURVE_H

#include <string>
#include <vector>

#include "UTModelBase.h"

////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
class UTModelYieldCurve : public UTModelBase
{
public:

    enum UT_InterpolateeType
    {
        UT_FWD_RATE = 0,
        UT_ZERO_RATE = 1,
        UT_MINUS_LOG_DF = 2
```

```
};

enum UT_InterpolationMethod
{
    UT_FLAT = 0,
    UT_CUBIC_SPLINE = 1,
    UT_LINEAR = 2
};

static std::string const ourClassTag;

// Destructor.
virtual ~UTModelYieldCurve();

// Defalut Constructor: construct the default flat yield curve at 3%
UTModelYieldCurve();

// Constructor with user specified flat rate
UTModelYieldCurve(double flatRate);

//Cinstructor with user specified yield curve
UTModelYieldCurve(
    const std::vector<double>&      timeLine,
    const std::vector<double>&      rates,
    UT_InterpolateeType interpolateeType = UT_FWD_RATE,
    UT_InterpolationMethod interpolationMethod = UT_FLAT);

//Clone
virtual UTModelBase* clone() const;

// Functions.
virtual std::string classTag() const { return ourClassTag; }

// Return the discount factor given a date
virtual double df( double time) const;
// Return the forward rate given two dates
virtual double forwardRate(double startTime, double endTime) const;
// Return minus log of DF
double lnDf(double time) const;

private:

// solver in the calibration only use this.
virtual void setComponent(unsigned int i, double component) { myRates[i] = component; }

//Yield Curve
std::vector<double> myTimeLine;
```

```
    std::vector<double> myRates;
    UT_InterpolateeType myInterpolatee;
    UT_InterpolationMethod myInterpolationMethod;
};
#endif // UT_MODEL_YIELD_CURVE_H
```

5.4 UTModelYieldCurve.cpp

```
#include "UTModelYieldCurve.h"

using namespace std;

////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
// Static data.
// The class tag
const string UTModelYieldCurve::ourClassTag = "Yield Curve Model";

////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
UTModelYieldCurve::~UTModelYieldCurve()
{
}

////////////////////////////////////////////////////////////////////////////
//Default constructor: Default yield curve of flat 3%
UTModelYieldCurve::UTModelYieldCurve()
    : UTModelBase(),
    myTimeLine(1, 1.0), myRates(1, 0.03), myInterpolatee(UT_FWD_RATE), myInterpolation-
Method(UT_FLAT)
{
}

////////////////////////////////////////////////////////////////////////////
UTModelYieldCurve::UTModelYieldCurve(double flatRate)
    : UTModelBase(),
    myTimeLine(1, 1.0), myRates(1, flatRate), myInterpolatee(UT_FWD_RATE), myInterpolation-
Method(UT_FLAT)
{
}

////////////////////////////////////////////////////////////////////////////
UTModelYieldCurve::UTModelYieldCurve(const vector<double>&      timeLine,
    const vector<double>&      rates,
```

```
    UT_InterpolateeType interpolateeType,
    UT_InterpolationMethod interpolationMethod)
    : UTModelBase(),
    myTimeLine(timeLine), myRates(rates), myInterpolatee(interpolateeType), myInterpolation-
Method(interpolationMethod)
{
}

//////////////////////////////////////////////////////////////////////////////
UTModelBase* UTModelYieldCurve::clone() const
{
    return new UTModelYieldCurve(*this);
}

//////////////////////////////////////////////////////////////////////////////
double UTModelYieldCurve::df(double time) const
{
    return exp(-1.0 * lnDf(time));
}

//////////////////////////////////////////////////////////////////////////////
double UTModelYieldCurve::forwardRate(double startTime, double endTime) const
{
    double rtn = df(startTime) / df(endTime) - 1.0;
    return rtn / (endTime - startTime);
}

//////////////////////////////////////////////////////////////////////////////
double UTModelYieldCurve::lnDf(double time) const
{
    //Assuming piecewise flat forward rate interpolation..
    if (time < 0.0)
    {
        return 0.0;
    }

    // Get the size of the time line.
    auto gridSize = myTimeLine.size();
    double previousTime = 0.0;
    double currentTime = 0.0;
    double sum = 0.0;
    unsigned int i = 0;
    for (i = 0; i < gridSize; ++i)
```

```
    {
        currentTime = myTimeLine[i];
        if (currentTime <= time)
        {
            sum += myRates[i] * (currentTime - previousTime);
            previousTime = currentTime;
        }
        else
        {
            //This means interpolation
            return sum + myRates[i] * (time - previousTime);
        }
    }

    //This means exterpolation (i.e., input time > last grid point of time)
    return sum + myRates[i-1] * (time - previousTime);
}
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
```

5.5 UTValuationEngineProduct.h

```
#ifndef UT_VALUATION_ENGINE_PRODUCT_H
#define UT_VALUATION_ENGINE_PRODUCT_H

#include <memory>
#include <vector>

// Forward declaration
class UTModelBase;
class UTModelYieldCurve;
class UTProductBase;
class UTProductLinearBase;
class UTProductCashflowBase;
class UTProductCashflowBullet;
class UTProductCashflowRateFixed;
class UTProductCashflowRateFloat;
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
class UTValuationEngineProductBase
{
public:
```

```
    // Destructor.
    virtual ~UTValuationEngineProductBase() {};
    // Constructor.
    UTValuationEngineProductBase(const UTModelBase & model);

    // Calculates the PV of the Product and accumulate it in the ResultPV object.
    virtual void calculatePV( double& result ) = 0;
    // Accessors
    const UTModelBase & modelBase() const { return myModelBase; }

protected:
    const UTModelBase & myModelBase;
};

////////////////////////////////////////////////////////////////////////////
class UTValuationEngineAnalyticProductLinear : public UTValuationEngineProductBase
{
public:

    // Destructor.
    virtual ~UTValuationEngineAnalyticProductLinear() {};
    // Constructor.
    UTValuationEngineAnalyticProductLinear(const UTModelBase & model, const UTProductLinearBase & product);

    // Accessor
    const UTProductLinearBase& linearProduct() const { return myProductLinear; }

    // Calculates the PV of the Product and accumulate it in the ResultPV object.
    virtual void calculatePV(double& result);

private:
    // Private functions
    unsigned int size() const { return mySubValuationEngines.size(); }
    UTValuationEngineProductBase& subValuationEngine(unsigned int i) { return *(mySubValuationEngines[i]); }
    const UTValuationEngineProductBase& subValuationEngine(unsigned int i) const { return *(mySubValuationEngines[i]); }

    const UTProductLinearBase & myProductLinear;
    // This valuation engine contains lots of 'smaller' valuation engines: one for each (undetermined) sub-product in the product linear
    std::vector<std::unique_ptr<UTValuationEngineProductBase> > mySubValuationEngines;
};

////////////////////////////////////////////////////////////////////////////
```

```
class UTValuationEngineProductCashflowBase : public UTValuationEngineProductBase
{
public:

    // Destructor.
    virtual ~UTValuationEngineProductCashflowBase() {};
    // Constructor.
    UTValuationEngineProductCashflowBase(const UTModelBase & model, const UTProductCash-
flowBase & product);

protected:

    // Accessors
    const UTProductCashflowBase & productCashFlowBase() const { return myProductCashFlow-
Base; }

    //member
   double myValue;
private:
    const UTProductCashflowBase & myProductCashFlowBase;
};

////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
// A yield curve model is used to value the following cashflows.
//
// UTProductCashflowBullet
// UTProductCashflowRateFixed
// UTProductCashflowRateFloat

class UTValuationEngineAnalyticYieldCurveProductCashflowBullet : public UTValuationEnginePro-
ductCashflowBase
{
public:

    // Destructor.
    virtual ~UTValuationEngineAnalyticYieldCurveProductCashflowBullet() {}
    // Constructor.
    UTValuationEngineAnalyticYieldCurveProductCashflowBullet(
        const UTModelYieldCurve & model,
        const UTProductCashflowBullet & cashFlow);
private:
    // References to the model and the cashflow.
    const UTModelYieldCurve & myModel;
```

```
    const UTProductCashflowBullet & myProductCashflow;

    // Calculated values.
    double myPayment;
    double myPaymentDf;
};
////////////////////////////////////////////////////////////////////////////
class UTValuationEngineAnalyticYieldCurveProductCashflowRateFixed : public UTValuation-
nEngineProductCashflowBase
{
public:

    // Destructor.
    virtual ~UTValuationEngineAnalyticYieldCurveProductCashflowRateFixed() {}
    // Constructor.
    UTValuationEngineAnalyticYieldCurveProductCashflowRateFixed(
        const UTModelYieldCurve & model,
        const UTProductCashflowRateFixed & cashFlow);

private:

    // References to the model and the cashflow.
    const UTModelYieldCurve & myModel;
    const UTProductCashflowRateFixed & myProductCashflow;

    // Calculated values.
    double myFlowPayment;
    double myFlowPaymentDf;
    double myRate;
};

////////////////////////////////////////////////////////////////////////////
class UTValuationEngineAnalyticYieldCurveProductCashflowRateFloat : public UTValuationEngine-
ProductCashflowBase
{
public:

    // Destructor.
    virtual ~UTValuationEngineAnalyticYieldCurveProductCashflowRateFloat() {}
    // Constructor.
    UTValuationEngineAnalyticYieldCurveProductCashflowRateFloat(
        const UTModelYieldCurve & model,
        const UTProductCashflowRateFloat & cashFlow);

private:
```

```
    // References to the model and the cashflow.
    const UTModelYieldCurve & myModel;
    const UTProductCashflowRateFloat & myProductCashflow;

    // Calculated values.
    double myFlowPayment;
    double myFlowPaymentDf;
    double myFlowPaymentWithZeroSpread;
    double myFlowValueWithZeroSpread;
    double myForwardRate;
};

////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
#endif // UT_VALUATION_ENGINE_PRODUCT_H
```

5.6 UTValuationEngineProduct.cpp

```
#include "UTEnum.h"
#include "UTValuationEngineProduct.h"
#include "UTValuationEngineFactory.h"
#include "UTProductCashflow.h"
#include "UTProductSwap.h"
#include "UTModelYieldCurve.h"

using namespace std;

////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
//UTValuationEngineProductBase
//
UTValuationEngineProductBase::UTValuationEngineProductBase(const UTModelBase & model)
    : myModelBase(model)
{
}

////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
//UTValuationEngineProductCashflowBase
//
UTValuationEngineProductCashflowBase::UTValuationEngineProductCashflowBase(
    const UTModelBase & model,
    const UTProductCashflowBase & product)
    :
```

```
    UTValuationEngineProductBase(model),
    myProductCashFlowBase(product)
{
}
////////////////////////////////////////////////////////////////////////////
void UTValuationEngineProductCashflowBase::calculatePV( double& resultPv )
{
    // Accumulate only Pv
    resultPv += myValue;
}
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
//UTValuationEngineAnalytivYieldCurveProductLinear
//
UTValuationEngineAnalyticProductLinear::UTValuationEngineAnalyticProductLinear(const UT-
ModelBase& model, const UTProductLinearBase & product)
    :
    UTValuationEngineProductBase(model),
    myProductLinear(product)
{
    unsigned long iSize = myProductLinear.size();
    mySubValuationEngines.resize(myProductLinear.size());
    for (unsigned long i = 0; i < myProductLinear.size(); ++i)
        mySubValuationEngines[i] = unique_ptr<UTValuationEngineProductBase>(UTValuationEngineFactory::new
*myProductLinear.underlying(i)));
}
////////////////////////////////////////////////////////////////////////////
// Calculates the PV of the Product and accumulate it
void
UTValuationEngineAnalyticProductLinear::calculatePV(double& resultPv)
{
    // Loops though each sub-valuation engine to accumulate the PV of each component or leg (or
sub-product...)
    for (unsigned int i = 0; i < size(); ++i)
    {
        subValuationEngine(i).calculatePV(resultPv);
    }
}
////////////////////////////////////////////////////////////////////////////
```

```
////////////////////////////////////////////////////////////////////////////
// UTValuationEngineAnalyticYieldCurveProductCashflowBullet
//
UTValuationEngineAnalyticYieldCurveProductCashflowBullet::UTValuationEngineAnalyticYieldCurveProductCashfl
    const UTModelYieldCurve & model,
    const UTProductCashflowBullet & cashFlow)
    : UTValuationEngineProductCashflowBase(model, cashFlow),
    myModel(model),
    myProductCashflow(cashFlow)
{
    // The payment date and payment.
    double payTime = myProductCashflow.paymentTime();
    myPayment = myProductCashflow.amount();

    // The value is signed by whether we pay (-) or receive (+).
    if (myProductCashflow.payReceive() == UT_PayReceive::UT_PAY)
    {
        myPayment *= -1.0;
    }

    // Check the payment date is after the value date.
    if (payTime >= 0.0)
    {
        // Get the discount factor from the curve.
        myPaymentDf = myModel.df(payTime);
        myValue = myPayment * myPaymentDf;
    }
    else
    {
        myPaymentDf = 1.0;
        myValue = 0.0;
    }
}

////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
//
// UTValuationEngineAnalyticYieldCurveProductCashflowRateFixed
//
UTValuationEngineAnalyticYieldCurveProductCashflowRateFixed::UTValuationEngineAnalyticYieldCurveProductC
    const UTModelYieldCurve & model,
    const UTProductCashflowRateFixed & cashFlow)
```

```
    : UTValuationEngineProductCashflowBase(model, cashFlow),
    myModel(model),
    myProductCashflow(cashFlow)
{
    // The rate is just a fixed coupon.
    myRate = myProductCashflow.coupon();

    // The payment
    const double paymentTime = myProductCashflow.paymentTime();
    myFlowPayment = myProductCashflow.notional() * myProductCashflow.accrued() * myRate;

    // The value is signed by whether we pay (-) or receive (+).
    if (myProductCashflow.payReceive() == UT_PayReceive::UT_PAY)
    {
        myFlowPayment *= -1.0;
    }

    // Check the payment date is after the value date.
    if (paymentTime >=0.0)
    {
        // Get the discount factor from the curve.
        myFlowPaymentDf = myModel.df(paymentTime);
        myValue = myFlowPayment * myFlowPaymentDf;
    }
    else
    {
        myFlowPaymentDf = 1.0;
        myValue = 0.0;
    }
}
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
//
// UTValuationEngineAnalyticYieldCurveProductCashflowRateFloat
//
UTValuationEngineAnalyticYieldCurveProductCashflowRateFloat::UTValuationEngineAnalyticYieldCurveProductCa
    const UTModelYieldCurve & model,
    const UTProductCashflowRateFloat & cashFlow)
    :
    UTValuationEngineProductCashflowBase(model, cashFlow),
    myModel(model),
    myProductCashflow(cashFlow)
```

```
{
    const double paymentTime = myProductCashflow.paymentTime();

    // If the payment date is STRICTLY before the model value date, do NOT take it into account
    if (paymentTime <
    {
        myFlowPaymentDf = 1.0;
        myFlowPayment = 0.0;
        myFlowPaymentWithZeroSpread = 0.0;
        myForwardRate = 0.0;
    }
    else
        // The payment date is on or after the value date, take the payment into account
    {
        // Calculate the (forward) value of the libor index (payment forward measure).
        myForwardRate = myModel.forwardRate(myProductCashflow.startTime(), myProduct-
Cashflow.endTime());
        // The payment, and its 'derivatives' w.r.t. the spread.
        double adjustedNotional = myProductCashflow.notional() * myProductCashflow.accrued();
        myFlowPayment = adjustedNotional * (myForwardRate + myProductCashflow.spread());
        myFlowPaymentWithZeroSpread = adjustedNotional * myForwardRate;

        // Get the discount factor from the curve.
        myFlowPaymentDf = myModel.df(paymentTime);
    }

    // The value is signed by whether we pay (-) or receive (+).
    if (myProductCashflow.payReceive() == UT_PayReceive::UT_PAY)
    {
        myFlowPayment *= -1.0;
        myFlowPaymentWithZeroSpread *= -1.0;
    }

    myValue = myFlowPayment * myFlowPaymentDf;
    myFlowValueWithZeroSpread = myFlowPaymentWithZeroSpread * myFlowPaymentDf;
}
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
```

5.7 UTValuationEngineFactory.h

```
#ifndef UT_VALUATION_ENGINE_FACTORY_H
#define UT_VALUATION_ENGINE_FACTORY_H
```

```
#include <memory>
#include <vector>

#include "UTValuationEngineProduct.h"

////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////
// Create the appropriate valuation engine by model, prodcut and method
class UTValuationEngineFactory
{
public:
    // Generic Valuation Engine for analytic method
    static std::unique_ptr<UTValuationEngineProductBase> newValuationEngineAnalytic(const UTModelBase& model, const UTProductBase& product, bool bThrow = false);

    // Valuation Engine for Analytic + YieldCurveModel
    static std::unique_ptr<UTValuationEngineProductBase> newValuationEngineAnalyticYieldCurve(const UTModelYieldCurve& model, const UTProductBase& product, bool bThrow = false);

};

#endif // UT_VALUATION_ENGINE_FACTORY_H
```

5.8 UTValuationEngineFactory.cpp

```
#include "UTValuationEngineFactory.h"
#include "UTValuationEngineProduct.h"
#include "UTProductCashflow.h"
#include "UTProductSwap.h"
#include "UTModelYieldCurve.h"

using namespace std;

////////////////////////////////////////////////////////////////////////
unique_ptr<UTValuationEngineProductBase> UTValuationEngineFactory::newValuationEngineAnalytic(const UTModelBase& model, const UTProductBase& product, bool bThrow)
{
    if (model.classTag() == "Yield Curve Model")
    {
        return newValuationEngineAnalyticYieldCurve(dynamic_cast<const UTModelYieldCurve&>(model), product, bThrow);
    }
    else
    {
        if (bThrow)
```

```
            {
                throw runtime_error("UTValuationEngineRegistry::The input model cannnot value the
product analytically.");
            }
        }

        return nullptr;
    }

    ////////////////////////////////////////////////////////////////////////////
    unique_ptr<UTValuationEngineProductBase> UTValuationEngineFactory::newValuationEngineAnalyticYieldCurve(c
UTModelYieldCurve& model, const UTProductBase& product, bool bThrow)
    {
        unique_ptr<UTValuationEngineProductBase> pValuationEngine(nullptr);
        if (product.classTag() == "Vanilla Leg")
        {
            pValuationEngine.reset(new      UTValuationEngineAnalyticProductLinear(model,      dy-
namic_cast<const UTProductLinearBase&>(product)));
        }
        else if (product.classTag() == "Vanilla Swap")
        {
            pValuationEngine.reset(new      UTValuationEngineAnalyticProductLinear(model,      dy-
namic_cast<const UTProductLinearBase&>(product)));
        }
        else if (product.classTag() == "Product Cashflow Bullet")
        {
            pValuationEngine.reset(new    UTValuationEngineAnalyticYieldCurveProductCashflowBul-
let(model, dynamic_cast<const UTProductCashflowBullet&>(product)));
        }
        else if (product.classTag() == "Product Cashflow Rate Fixed")
        {
            pValuationEngine.reset(new  UTValuationEngineAnalyticYieldCurveProductCashflowRate-
Fixed(model, dynamic_cast<const UTProductCashflowRateFixed&> (product)));
        }
        else if (product.classTag() == "Product Cashflow Rate Float")
        {
            pValuationEngine.reset(new  UTValuationEngineAnalyticYieldCurveProductCashflowRate-
Float(model, dynamic_cast<const UTProductCashflowRateFloat&> (product)));
        }
        else
        {
```

```
            if (bThrow)
            {
                throw runtime_error("UTValuationEngineFactory::The input product cannnot be valued by Yield curve model analytically.");
            }
        }

        return pValuationEngine;
    }

    ////////////////////////////////////////////////////////////////////////////
    ////////////////////////////////////////////////////////////////////////////
```

5.9 UTModelFactory.h

```
    #ifndef UT_MODEL_FACTORY_H
    #define UT_MODEL_FACTORY_H

    #include <vector>

    #include "UTModelYieldCurve.h"
    #include "UTProductCashflow.h"
    #include "UTBisection.h"

    ////////////////////////////////////////////////////////////////////////////
    ////////////////////////////////////////////////////////////////////////////
    // UTModelFactory
    //
    class UTModelFactory
    {
    public:
        // Destructor.
        virtual ~UTModelFactory() {}
        // Constructor.
        UTModelFactory() {}

        // Creates a 'calibrated' yield curve model
        static std::unique_ptr<UTModelYieldCurve> newModelYieldCurve(
            const std::vector<double> & swapMaturities,
            const std::vector<double> & swapRates,
            const UTModelYieldCurve::UT_InterpolationMethod& interpMethod = UTModelYield-
Curve::UT_FLAT,
            const UTModelYieldCurve::UT_InterpolateeType& interpolateeType = UTModelYield-
Curve::UT_FWD_RATE );
```

```
};
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
//Helper Class for model calibration
class UTSolveForModelComponent : public UTBisection
{
    // Object for this class: Calibration of the given model
public:

    virtual ~UTSolveForModelComponent() {}
    UTSolveForModelComponent(UTModelBase &model, const UTProductBase& product, double
target, unsigned int componentNumber)
        : UTBisection(),
        myModel(model),
        myProduct(product),
        myTarget(target),
        myComponentNumber(componentNumber){}

    // Overloaded penalty function
    virtual double error (double x);

private:

    UTModelBase &      myModel;
    const UTProductBase& myProduct;
    double myTarget;
    unsigned int myComponentNumber;
};
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
#endif // UT_MODEL_FACTORY_H
```

5.10 UTModelFactory.cpp

```
#include <memory>

#include "UTEnum.h"
#include "UTModelFactory.h"
#include "UTProductSwap.h"
#include "UTValuationEngineFactory.h"
#include "UTBisection.h"

using namespace std;

////////////////////////////////////////////////////////////////////////////
```

```
////////////////////////////////////////////////////////////////////////////
unique_ptr<UTModelYieldCurve>
UTModelFactory::newModelYieldCurve(
const vector<double> & swapMaturities,
const vector<double> & swapRates,
const UTModelYieldCurve::UT_InterpolationMethod& interpMethod,
const UTModelYieldCurve::UT_InterpolateeType& interpolateeType)
{
    // Check the size of the 2 vectors
    if (swapMaturities.size() != swapRates.size())
    {
        throw runtime_error("UTModelFactory: The size of swap maturities and rates should be the
same.");
    }

    // Create a temporal yield curve model with inputed interp method
    unique_ptr<UTModelYieldCurve> pYieldCurveModel(new UTModelYieldCurve(swapMaturities,
swapRates, interpolateeType, interpMethod));

    //We are assuming that the swap maturities and rates are ordered correctly...
    for (unsigned int i = 0; i < swapMaturities.size(); ++i)
    {
        // create product.
        // Assuming the swap convention is like JPY swaps..
        UTProductSwapVanilla vanillaSwap(0, swapMaturities[i], swapRates[i], 0.5, 0.5, 10000.0,
UT_PayReceive:: UT_RECEIVE);
        UTSolveForModelComponent solver(*pYieldCurveModel, vanillaSwap, 0.0,i);
        double rate = solver.root(-0.1, 1.0);
    }

    // Calibration is done! So return the calibrated model
    return pYieldCurveModel;
}
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
double
UTSolveForModelComponent::error(double x)
{
    // Pricing
    myModel.setComponent(myComponentNumber,x);

    // Usually Calibration is done to analytic prices...
```

```
    unique_ptr<UTValuationEngineProductBase>pricer(UTValuationEngineFactory::newValuationEngineAnalytic(m
myProduct));

    double pv = 0.0;
    pricer->calculatePV(pv);
    return pv - myTarget;
}
//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////
```

5.11 UTBisection.h

```
#ifndef UT_BISECTION_H
#define UT_BISECTION_H

#include <float.h>
#include <cmath>

//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////
// Polymorphism solution
class UTBisection
{
public:
    //
    // f(x) is the value of the function at abscissa x whose zeros we are looking for.
    // The error function must be overloaded.
    virtual double error(double x) = 0;
    //
    // Virtual destructor.
    virtual ~UTBisection(){}
    //
    // NOTE: this root finder only considers convergence in the abscissa, i.e. it iterates until further
progress in the argument
    // is smaller than absoluteAccuracy.
    //
    double root(double lowerBound, double upperBound, double absoluteAccuracy = 15 *
DBL_EPSILON, unsigned long maxIterations = (DBL_DIG * 10) / 3);
};
//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////
#endif // UT_BISECTION_H
```

5.12 UTBisection.cpp

```
#include <stdexcept>

#include "UTBisection.h"

using namespace std;

////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////
double UTBisection::root(double x1, double x2, double accuracy, unsigned long maxIterations)
{
    double f1 = error(x1);
    double f2 = error(x2);

    double xmid, fmid;
    for (unsigned long i = 0; i<maxIterations; ++i)
    {
        xmid = 0.5 * (x1 + x2);
        fmid = error(xmid);
        if (fmid == 0.0)
        {
            return xmid;
        }
        else if (f1*fmid < 0.0)
        {
            x2 = xmid;
            f2 = fmid;
        }
        else
        {
            x1 = xmid;
            f1 = fmid;
        }

        if (fabs(x2 - x1) <= accuracy || fmid == 0.0)
            return 0.5 * (x2 + x1);
    }

    throw runtime_error("UTBisection: Root search did not converge.");

    // Should never get here
    return 0.0;
}
```

5.13 UTTest.h

```
#ifndef UT_TEST_H
#define UT_TEST_H

void pricingTest();
void yieldCurveCalibration();
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
#endif // UT_NEWTON_H
```

5.14 UTTest.cpp

```
#include<iostream>
#include<fstream>
#include<string>

#include "UTEnum.h"
#include "UTProductSwap.h"
#include "UTModelYieldCurve.h"
#include "UTValuationEngineFactory.h"
#include "UTModelFactory.h"
#include "UTTest.h"

using namespace std;

void yieldCurveCalibration()
{
    vector<double> swapRates{ 0.01, 0.03, 0.05 };
    vector<double> swapMaturities{ 1.0, 3.0, 5.0 };

    //Model generation
    auto yieldCurve = UTModelFactory::newModelYieldCurve(swapMaturities, swapRates);

    // Products
    UTProductSwapVanilla vanillaSwap1(0, 1.0, 0.01, 0.5, 0.5, 10000.0, UT_PayReceive::UT_RECEIVE);
    UTProductSwapVanilla vanillaSwap2(0, 3.0, 0.03, 0.5, 0.5, 10000.0, UT_PayReceive::UT_RECEIVE);
    UTProductSwapVanilla vanillaSwap3(0, 5.0, 0.05, 0.5, 0.5, 10000.0, UT_PayReceive::UT_RECEIVE);

    //Pricing Test
    auto pricer1 = UTValuationEngineFactory::newValuationEngineAnalyticYieldCurve(*yieldCurve,
vanillaSwap1);
    double pv = 0.0;
    pricer1->calculatePV(pv);
```

```
        cout << "the PV of 1st swap is " << pv << ".\n";
        auto pricer2 = UTValuationEngineFactory::newValuationEngineAnalyticYieldCurve(*yieldCurve,
vanillaSwap2);
        pv = 0.0; //reset of pv
        pricer2->calculatePV(pv);
        cout << "the PV of the 2nd swap is " << pv << ".\n";
        auto pricer3 = UTValuationEngineFactory::newValuationEngineAnalyticYieldCurve(*yieldCurve,
vanillaSwap3);
        pv = 0.0;
        pricer3->calculatePV(pv);
        cout << "the PV of the 3rd swap is " << pv << ".\n";
    }
    void pricingTest()
    {
        // Products
        UTProductSwapVanilla vanillaSwap(0, 10, 0.03, 0.5,0.5, 10000.0, UT_PayReceive::UT_RECEIVE);
        UTProductLegVanilla fixedLeg(UT_FixedFloat::UT_FIXED, 0, 10, 0.03, 0.5, 10000.0,
UT_PayReceive::UT_RECEIVE);
        UTProductLegVanilla floatLeg(UT_FixedFloat::UT_FLOAT, 0, 10, 0.0, 0.5, 10000.0, UT_PayReceive::UT_PAY);
        //Model
        UTModelYieldCurve yieldCurve;
        //Pricing
        auto pricerFixedLeg(UTValuationEngineFactory::newValuationEngineAnalyticYieldCurve(yieldCurve,
fixedLeg));
        double pv = 0.0;
        pricerFixedLeg->calculatePV(pv);
        cout << "the PV of the fixed leg is " << pv << ".\n";
        auto pricerFloatLeg(UTValuationEngineFactory::newValuationEngineAnalyticYieldCurve(yieldCurve,
floatLeg));
        pv = 0.0; //reset of pv
        pricerFloatLeg->calculatePV(pv);
        cout << "the PV of the float leg is " << pv << ".\n";
        auto pricerSwap(UTValuationEngineFactory::newValuationEngineAnalyticYieldCurve(yieldCurve,
vanillaSwap));
        pv = 0.0;
        pricerSwap->calculatePV(pv);
        cout << "the PV of the swap is " << pv << ".\n";
    }
```

5.15 UTMain.cpp

```
#include<iostream>

#include "UTTest.h"
#include "UTProductSwap.h"
#include "UTModelYieldCurve.h"

using namespace std;

int main()
{
    try
    {
        //Put your test routine
        yieldCurveCalibration();
    }
    catch (runtime_error err)
    {
        cout << err.what();
    }

    // The final input to stop the routine
    double tmp;
    cin >> tmp;
    return 0;
}
```

著作権と免責事項