State processes and their role in design and implementation of financial models

Dmitry Kramkov

Carnegie Mellon University, Pittsburgh, USA

Implementing Derivative Valuation Models, FORC, Warwick, February 24

Outline

"Financial Computing with C++": a course in MSCF

Goals of the course:

"Theoretical": review and expand the knowledge of the basic topics:

1. Object Oriented Programming with C++
2. Arbitrage-Free Pricing of Derivatives
3. Stochastic Calculus
4. Numerical Analysis

"Practical": improve the ability to use C++ (**speak C++ !**) for practical financial computations.

Case study: `cfl` (**Library** for the course **Financial Computing**)

Final exam

- Students are given 3 hours to price 3 derivative securities.

In the final exam for 2005 students had to price:

1. "BOOST" (Banking on Overall Stability Option)
2. Ratchet Bond
3. Target Inverse Floater Swap

Results for Pittsburgh's section:

# of solved problems	3	2	1	0
# of students	3	9	1	12

"BOOST": Banking On Overall Stability

N : notional amount.

$L < U$: lower and upper barriers.

$(t_i)_{1 \le i \le m}$: barrier times

The option terminates at the first barrier time, when the price of the stock hits either of the barriers, that is, at the barrier time t_{i^*}, which index is given by

$$i^* = \min\{1 \le i \le m : \quad S(t_i) > U \text{ or } S(t_i) < L\}.$$

At the exit time t_{i^*} the holder of the option receives the payoff $N\frac{i^*-1}{m}$ (the product of the notional amount on the percentage of the barrier times that the price of the stock spends inside two barriers).

Ratchet bond

N : notional

c : initial coupon rate

d : reset value for the coupon rate ($d < c$).

δt : interval of time between the payments given as year fraction.

m : total number of coupon payments.

L : the redemption price of the bond as percentage of the notional. Typically, $L < 1$.

After coupon payment the issuer can reset the coupon rate from the original (higher) value c to the (lower) reset value d. However, later the holder can sell the bond back to the issuer for the redemption value LN.

Target Redemption Inverse Floater

δt : interval of time between the payments given as year fraction.

m : maximal number of payments

N : notional amount.

R : strike fixed rate.

Q : the total coupon paid to a client as percentage of the notional.

We pay *inverse float* coupon ($= N\delta t \max(R - \text{Libor}, 0)$) to the client and receive Libor until the total coupon reaches the threshold QN. Then the trade is terminated. Note that the total coupon paid to the client over the time of the trade equals exactly QN.

Design of pricing library

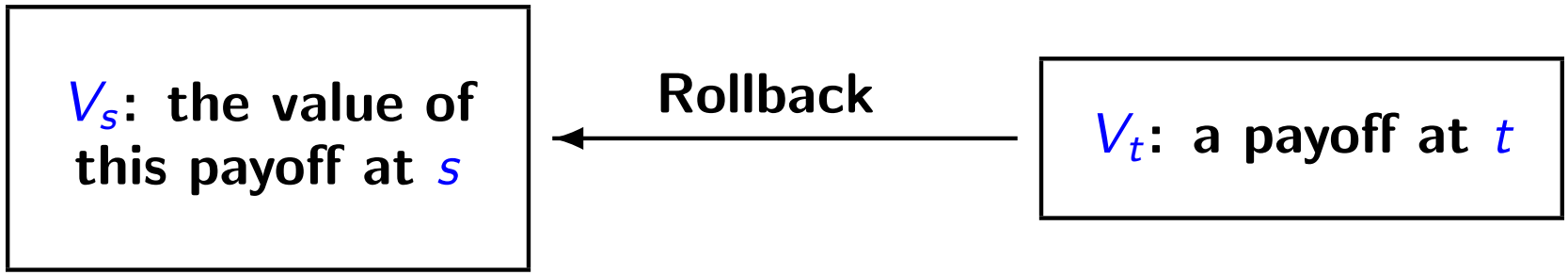
Models:
Black
...
Hull-White
...

Derivatives:
Boost option
...
Ratchet Bond
Target Redemption Inverse Floater
...

Basic goal of design: **re-usability**.

1. Tools of C++: inheritance, templates, ...
2. Basic concepts of Arbitrage-Free Pricing Theory: rollback operator, **state process**, ...

Rollback operator



Risk-neutral pricing:

$$V_s = \mathcal{R}_s[V_t] = \mathbb{E}_s^*[V_t \exp(-\int_s^t r_u du)]$$

where

(r_t): short-term interest rate

$\mathbb{P}^*$: (money market) martingale measure

State processes

Main Idea: efficient storage scheme for *relevant* random variables.

Remark

The storage scheme should be *adapted* to the *type* of the derivative security.

Definition

A process $(X_t)_{0 \leq t \leq T}$ is called a **state process** if

- for all $s < t$ and any deterministic function $f = f(x)$
- there is a deterministic function $g = g(x)$

such that

$$g(X_s) = \mathcal{R}_s[f(X_t)].$$

State and Markov processes

Theorem

The following conditions are equivalent:

1. X *is a state process*
2. *We have*
   - 2.1 X *is a Markov process under the money market martingale measure* $\mathbb{P}^*$
   - 2.2 *For any time* t *the short term interest rate* r_t *is determined by* X_t*, that is,*

$$r_t = h(X_t, t), \quad t > 0,$$

for some deterministic function $h = h(x, t)$

Some standard examples

1. Black and Scholes model:

$$dS_t = S_t(q_t dt + \sigma_t dW_t),$$

   Here (S_t) is a state process.

2. Commodity model with mean-reversion:

$$dS_t = S_t\left[(\theta_t - \lambda_t \ln S_t)dt + \sigma_t dW_t\right]$$

   Here (S_t) is a state process.

3. Hull and White model for interest rates:

$$dr_t = (\theta_t - \lambda_t r_t)dt + \sigma_t dW_t$$

   Here (r_t) is a state process.

"Implementation" of a financial model

1. The specification of a state process X (the choice of the state process is determined by the type of derivative security).
2. The implementation of the following operations for random variables from the families

$$\{f(X_t):\ f = f(x)\},\ t > 0$$

   2.1 For given time t: all arithmetic and functional operations
   2.2 Between two times $s < t$: rollback operator.

Example

Standard implementation of Black and Scholes model allows us to operate at time t with random variables of the form

$$\{f(S_t):\ f = f(x)\}$$

A model in `cfl` library

Basic components:

1. $(t_i)_{0 \le i \le M}$: *sorted* vector of **event times** given as year fractions.

   Event times: all times needed to price a *particular* derivative security (exericse times, barrier times, reset times, ...).

   Numerical efficiency: create the vector of event times with a *smallest* size.

2. $X = (X^0, \ldots, X^{d-1})$: ($d$-dimensional) **state process**.

At an event time t_i we operate with random variables:

$$\mathcal{X}_{t_i} = \{f(X_{t_i}) : \quad f = f(x)\}$$

represented by the class `cfl::Slice`

cfl::Slice

There are 2 types of operations for `cfl::Slice`:

1. At given event time t_i: all possible arithmetic, functional, etc. . For example, if

   `uSpot:` `cfl::Slice` for the spot price $S(t_i)$ at t_i

   `dK:` `double` for a cash amount K at t_i

   then

   ```
   Slice uCall = max(uSpot - dK, 0.);
   ```

   creates `cfl::Slice` for the payoff

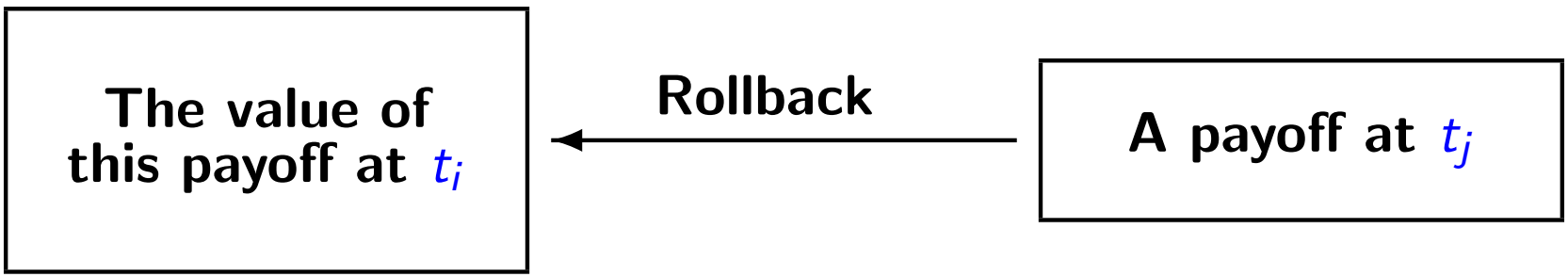
   $$\max(S(t_i) - K, 0)$$

   of the call option with strike K and maturity t_i.

cfl::Slice

There are 2 types of operations for `cfl::Slice`:

2. Between two event times $t_i < t_j$: only **rollback** operator.



Algorithm for pricing of standard call:

```
...
//two event times:  0 (initial) and 1 (maturity)
Slice uCall = max(uModel.spot(1) - dK, 0);
uCall.rollback(0);
```

Program flow

1. Basic objects of the type `cfl::Slice` such as
   1.1 spot prices
   1.2 discount factors, etc.

   are created by an implementation of a particular financial model
2. We then manipulate these basic objects using the provided operators and functions:
   2.1 for given event time: all arithmetic and functional operations;
   2.2 between two event times: rollback operator.

Code for BOOST (Banking on Overall Stability) option

```
...
  Model uModel(rData, uEventTimes, dInterval, dQuality);
  int iTime = uModel.eventTimes().size()-1;
  Slice uOption = uModel.cash(iTime, dNotional);
  while (iTime > 0) {
    //uOption = value to continue
    Slice uInd = indicator(uModel.spot(iTime), dLowerBarrier)*
      indicator(dUpperBarrier, uModel.spot(iTime));
    uOption *= uInd;
    double dPayoff = dNotional*(iTime-1.)/rBarrierTimes.size();
    uOption += dPayoff*(1. - uInd);
    iTime--;
    uOption.rollback(iTime);
  }
...
```

Code for Ratchet Bond

```
...
Model uModel(rData, uTimes, dInterval, dQuality);
int iTime = uTimes.size()-1; //last minus one coupon time
double dOriginalCoupon =  dNotional * rBond.rate * dPeriod;
double dResetCoupon = dNotional * dResetCouponRate * dPeriod;
double dRedemptValue = dRedemptionPrice * dNotional;
Slice uBondBeforeReset = uModel.discount(iTime,dMaturity) *
  (dNotional + dOriginalCoupon);
Slice uBondAfterReset = uModel.discount(iTime,dMaturity) *
  (dNotional + dResetCoupon);
while (iTime > 0) {
  uBondAfterReset = max(uBondAfterReset, dRedemptValue);
  uBondBeforeReset = min(uBondAfterReset, uBondBeforeReset);
  uBondAfterReset+= dResetCoupon;
  uBondBeforeReset+=dOriginalCoupon;
  iTime--;
  uBondBeforeReset.rollback(iTime);
  uBondAfterReset.rollback(iTime);
 }
...
```

Pricing of path-dependent derivatives

Assume that we have **standard** implementation of some interest rate model, that is, at any time t we can work with random variables:

$$\{f(B(t,T)) : f = f(x) \quad T > t\}$$

where $B(t,T)$ is a discount factor with maturity T.

- Using this implementation we can price different standard and barrier, European and American options.
- However, we are not able to handle **Path-Dependent** derivatives such as *Target Redemption Inverse Floater*.

Solution: extend the dimension of the model by adding new component Y to the original state process.

General framework

Assume that we are given an "implementation" of a financial model corresponding to a particular choice of a state process X, that is, for random variables from the sets

$$\mathcal{X}_t = \{f(X_t) : \ f = f(x)\}, \ t > 0$$

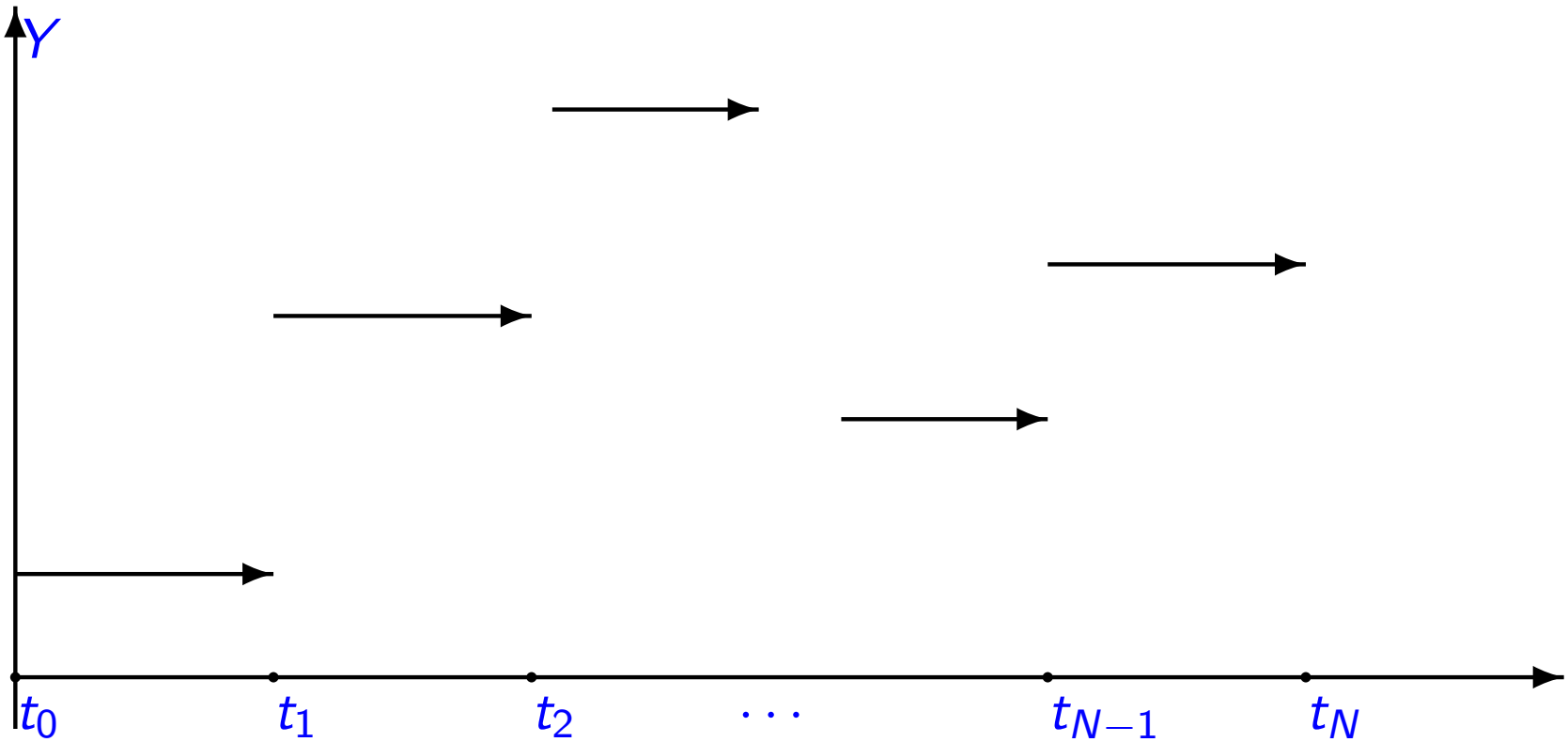
the following operations are implemented:

1. for given time t — all arithmetic and functional
2. between two times $s < t$ — rollback, that is for any $f = f(x)$ we know how to compute $g = g(x)$ such that

$$g(X_s) = \mathcal{R}_s(f(X_t))$$

General framework

Consider also a stochastic process Y which values change at **reset times** $t_1, \ldots, t_N$:



Main Theorem on Path-Dependence

Question: is (X, Y) a state process?

Theorem

Assume that for any **reset time** t_{i+1} *there is a deterministic function* $G_{i+1} = G_{i+1}(x, y)$ *(***reset function***) such that*

$$Y_{t_{i+1}} = G_{i+1}(X_{t_{i+1}}, Y_{t_i})$$

Then (X, Y) *is a state process.*

Remark

The value of Y at a reset time t is determined by

- the value of the **original** state process X **at** t
- and the value of Y **before** t.

Implementation in `cfl` library

1. We start with "standard" implementation of the model determined by the **basic** state process X.
2. To price a path dependent derivative security we add another state process Y determined by
   - 2.1 **reset times:** $t_1, \ldots, t_N$
   - 2.2 **reset functions:** $(G_i)_{1 \leq i \leq N}$

$$Y_{t_{i+1}} = G_{i+1}(X_{t_{i+1}}, Y_{t_i}).$$

3. Classes for path dependent processes:
   - 3.1 Interface class `cfl::IResetValues` (describes reset functions).
   - 3.2 Concrete class `cfl::PathDependent` (is related to `cfl::IResetValues` through **pimpl** idiom).

Code for Target Redemption Inverse Floater

```
class TotalNextCoupon: public IResetValues
{
public:
  TotalNextCoupon(const Model & rModel, double dCapRate, double dPeriod)
    :m_dCapRate(dCapRate), m_dPeriod(dPeriod), m_rModel(rModel)
  {}
  Slice resetValues(unsigned iTime, double dBeforeReset) const
  {
    return dBeforeReset + m_dPeriod *
      max(m_dCapRate - rate(m_rModel, iTime, m_dPeriod),0.);
  }
private:
  const Model & m_rModel;
  double m_dCapRate, m_dPeriod;
};


PathDependent totalNextCoupon(const Model & rModel,
     const std::vector<unsigned> & rResetIndexes,
     double dCapRate, double dPeriod)
{
  return PathDependent(new TotalNextCoupon(rModel, dCapRate, dPeriod),
       rResetIndexes, 0., 0.);
}
```

Code for Target Redemption Inverse Floater

```
...
Model uModel(rData, uTimes, dInterval, dQuality); //standard model

std::vector<unsigned> uResetIndexes(uTimes.size(),0);
std::transform(uResetIndexes.begin(), uResetIndexes.end()-1,
       uResetIndexes.begin()+1, std::bind1st(std::plus<unsigned>(),1));
unsigned iState =
  uModel.addState(totalNextCoupon(uModel,uResetIndexes,dCapRate,dPeriod));
//extended model

int iTime = uTimes.size()-1; //last minus one payment
Slice uSwap = uModel.cash(iTime, 0.);
while (iTime >= 0) {
  //uSwap = current value of all payments after the next payment time
  Slice uNextCoupon = max(dCapRate-rate(uModel,iTime,dPeriod),0)*dPeriod;
  Slice uTotalNextCoupon = uModel.state(iTime, iState);
  Slice uTotalCouponToday = uTotalNextCoupon - uNextCoupon;
  Slice uIndContinueNextTime = indicator(dMaxCoupon, uTotalNextCoupon);
  if (iTime == uTimes.size()-1) {
    uIndContinueNextTime = uModel.cash(iTime, 0.);
  }
  uNextCoupon *= uIndContinueNextTime;
  uNextCoupon += (1. - uIndContinueNextTime)*(dMaxCoupon-uTotalCouponToday);
  Slice uIndContinueToday = indicator(dMaxCoupon, uTotalCouponToday);
  double dNextPaymentTime = uModel.eventTimes()[iTime] + dPeriod;
  Slice uDiscount = uModel.discount(iTime, dNextPaymentTime);
  uSwap -= uIndContinueToday*(uNextCoupon*uDiscount - (1. - uDiscount));
  iTime--;
  if (iTime >=0) {
    uSwap.rollback(iTime);
  }
 }
uSwap *= rSwap.notional;
...
```

Models with identical state process

Consider two financial models A and B such that

- They have the same state process X.
- The model A has been implemented for the state process X ("old" model)
- The model B is "new".

Goal: implement B in terms of A.

Main difficulty: implement $\mathcal{R}^B$ in terms of $\mathcal{R}^A$.

Rollback density

Definition

We call $Z = (Z_t)$ a **rollback density** of B with respect to A (notation: $Z = \frac{d\mathcal{R}^B}{d\mathcal{R}^A}$) if for any $s < t$ and any payoff ξ at t

$$\mathcal{R}_s^B[\xi] = \frac{1}{Z_s}\mathcal{R}_s^A[Z_t\xi]$$

Remark

The concept of **rollback density** for two financial models is closely related to the concept of **Radon-Nikodym** derivative for two probability measures.

Rollback density

Denote

$d^A(s,T)$: discount factor in model A for maturity T computed at $s < T$.

$d^B(s,T)$: discount factor in model B for maturity T computed at $s < T$.

Theorem

A rollback density process of the model B with respect to the model A is given by:

$$Z_s = \frac{d\mathcal{R}_s^B}{d\mathcal{R}_s^A} = \frac{d^A(s,T)}{d^B(s,T)}.$$

Rollback density for "similar" models

If the models A and B share the same state process X then for any $s > 0$ there are deterministic functions $f_s = f_s(x)$ and $g_s = g_s(x)$ such that

$$\begin{aligned} d^A(s,T) &= f_s(X_s) \\ d^B(s,T) &= g_s(X_s) \end{aligned}$$

It follows that

$$Z_s = \frac{d\mathcal{R}_s^B}{d\mathcal{R}_s^A} = \frac{d^A(s,T)}{d^B(s,T)} = \frac{f_s(X_s)}{g_s(X_s)} = h_s(X_s)$$

Implementation of "similar" models

Therefore, given the implementation of the model A associated with the state process X we can easily provide the implementation of the model B for the same state process.
Indeed, if

$$\xi = \phi(X_t)$$

is a payoff at time t ($\phi = \phi(x)$), then

$$\mathcal{R}_s^B[\phi(X_t)] = \frac{1}{h_s(X_s)} \mathcal{R}_s^A[h_t(X_t)\phi(X_t)]$$

Key example: Brownian motion

A popular choice of a state process for many one-factor models is

$$X_t = \int_0^t \sigma(u) dW_u$$

where

$\sigma = \sigma(t)$: deterministic **volatility**

$W = (W_t)_{t \geq 0}$: standard **Brownian motion**

This process appears, for example, in

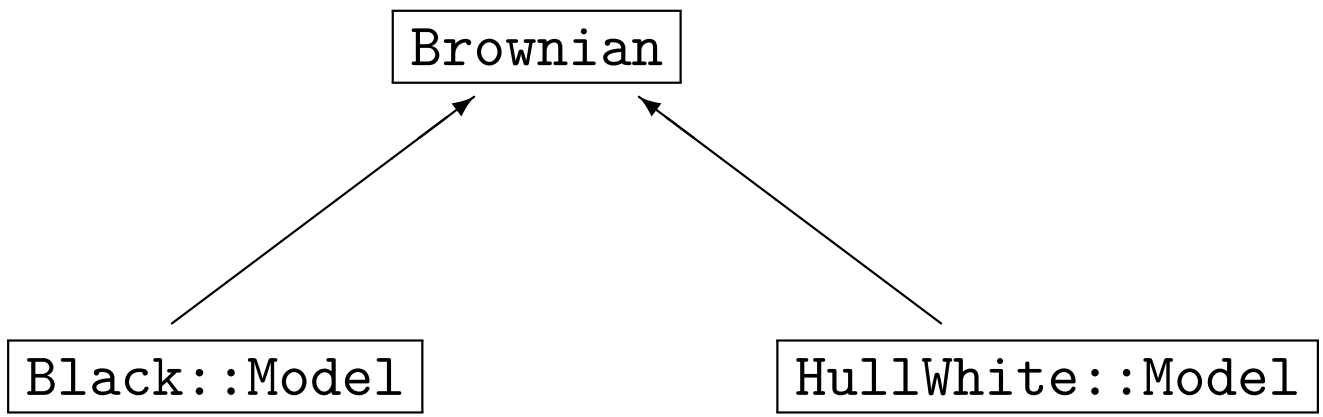
1. Black model
2. Hull and White model
3. Black-Karachinski model
4. Black-Derman-Toy model etc..

Brownian model in `cfl`

In `cfl` an "artificial" **Brownian** model has been defined, where interest rate is 0 and, hence,

$$\mathcal{R}_s[\cdot] = \mathbb{E}_s[\cdot].$$

This model has been used then to implement Black and Hull-White models. This is **great for testing**!



Multi-factor FX model

Problem: price FX swaption where domestic currency pays fixed and foreign currency pays float.
We need 2 factors:

1. FX rate S
2. Domestic short term interest rate r

"Naive" simplest model: (Black + Hull-White)

$$dS_t = S_t((r_t - q)dt + \sigma dW_t)$$
$$dr_t = (\theta(t) - \lambda r_t)dt + \kappa dB_t$$

where B and W are standard Brownian motions with *constant* correlation ρ:

$$\rho = \frac{\langle B, W \rangle_t}{\sqrt{\langle B, B \rangle_t}\sqrt{\langle W, W \rangle_t}} (= \frac{dBdW}{dt})$$

Multi-factor FX model

Looks fine. However, one can show that in this case state processes are given by

$$X_t = W_t \quad Y_t = \int_0^t e^{\lambda s} dB_s$$

The correlation coefficient between X and Y is *time-dependent*:

$$\frac{\langle X, Y \rangle_t}{\sqrt{\langle X, X \rangle_t}\sqrt{\langle Y, Y \rangle_t}} = \rho \frac{\int_0^t e^{\lambda u} du}{\sqrt{\int_0^t e^{2\lambda u} du}\sqrt{t}} \neq \text{const.}$$

Hence, we can not choose state processes to be two *independent* Brownian motions.

Multi-factor FX model

"Better" model:

$$dS_t = S_t((r_t - q)dt + \sigma e^{\lambda t} dW_t)$$
$$dr_t = (\theta(t) - \lambda r_t)dt + \kappa dB_t$$

In this case state processes are given by

$$X_t = \int_0^t e^{\lambda s} dW_s \quad Y_t = \int_0^t e^{\lambda s} dB_s$$

and

$$\frac{\langle X, Y \rangle_t}{\sqrt{\langle X, X \rangle_t}\sqrt{\langle Y, Y \rangle_t}} = \rho = \text{const}$$

Easy to implement as we can choose state processes to be two **independent Brownian motions**.

Summary

The concept of **state process** facilitates greatly the building of powerful object-oriented pricing libraries:

- Centerpiece of design
- Elegant implementation of path dependent derivatives
- Cross model implementation
- Important role in selection of "right" financial models.