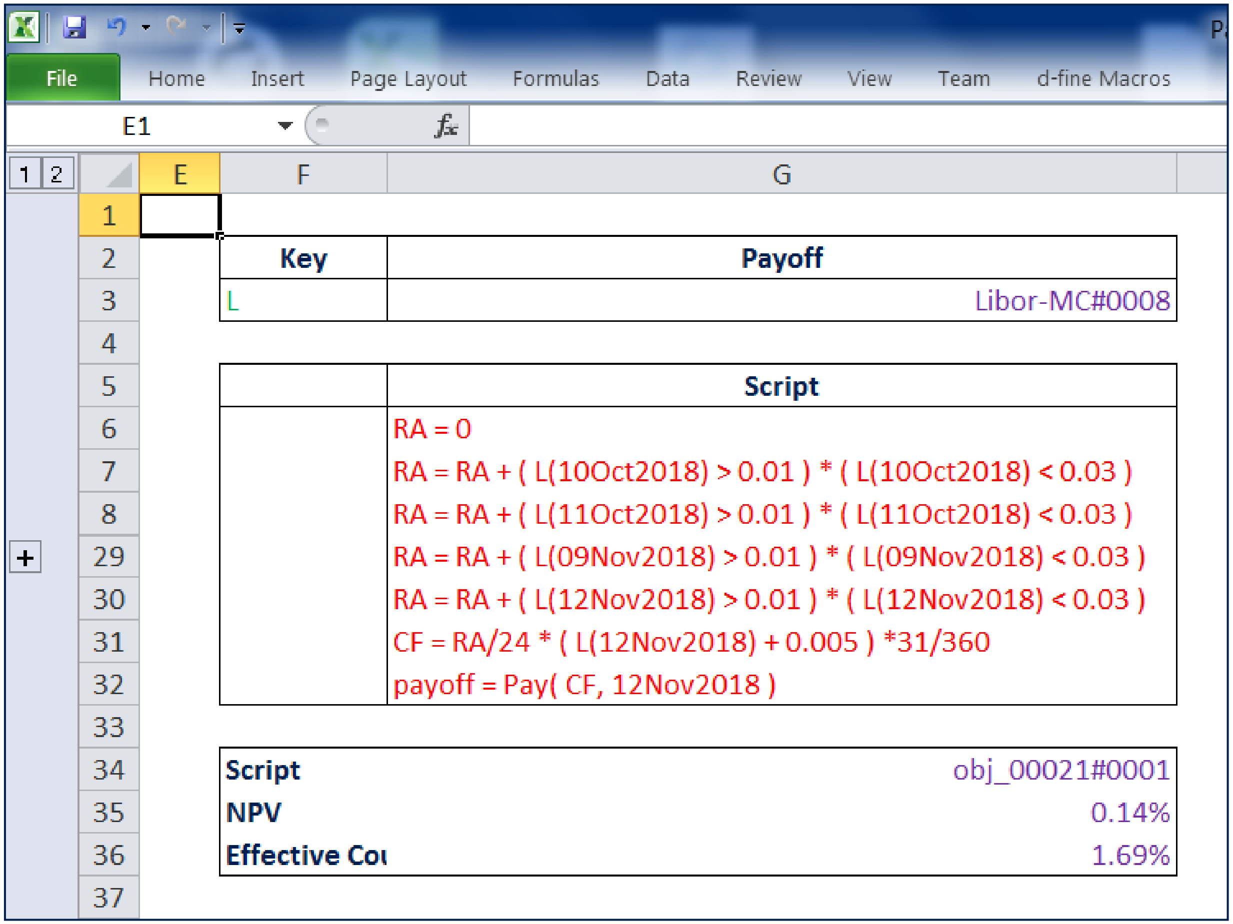
Structured Payoff Scripting in QuantLib

Dr Sebastian Schlenkrich

Dusseldorf, November 30, 2017

Why do we want a payoff scripting language? Let's start with a teaser example…



Payoff scripting provides great flexibility to the user and quick turnaround for ad-hoc analysis

d-fine

Agenda

» Payoffs, Paths and Simulations

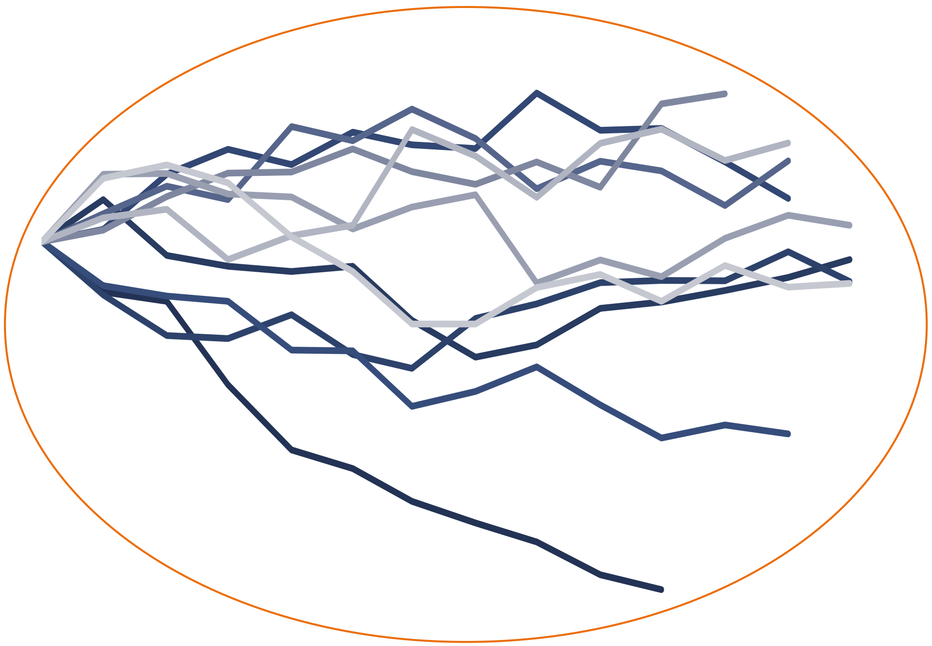
» A Flex/Bison-based Parser for a Bespoke Scripting Language

» Some Scripting Examples

» Summary

Payoffs, Paths and Simulations

A path is an abstract representation of the evolution of the world in time



General Path

$$p: [0, +\infty) \rightarrow \mathbb{R}^N$$

Alternatives/specialisations:

» 1-factor modells on **discrete observation** dates

$$p = [p_0, \dots, p_M] \in \mathbb{R}^M$$

» 1-factor model for **European payoffs**

$$p = p_0 \in \mathbb{R}$$

Payoff allows calculating a scalar quantity for a particular evolution (or realisation) of the world

$$V: p \mapsto \mathbb{R}$$

We consider general (abstract) paths and payoffs as functions mapping a path to a scalar quantity

Why does it have to be that abstract?

Assume $p = [p_0, \ldots, p_M] \in \mathbb{R}^M$ then a payoff is a functional $V: \mathbb{R}^M \to \mathbb{R}$

- In C++ this may just be any function with the signature `double payoff(vector<double> p)`
- Example **European call option**

```
double call(vector<double> p) {
    double strike = /* obtained from script context */
    return max(p.back()-strike,0);
}
```

- Such functions could be created dynamically, e.g. via C++ integration of other languages[1], e.g.
  - JNI + Scala for scripting in Scala
  - RInside for scripting in R

But what if the model and thus the interpretation of p changes?

- Model A: $p_i = S(t_i)$ (direct asset modelling)
- Model B: $p_i = \log(S(t_i))$ (log-asset modelling)

The payoff should not know what *kind of* the path is. Instead the payoff should only use a pre-defined interface to derive its value

(1) for details see e.g. hpcquantlib.wordpress.com/2011/09/01/using-scala-for-payoff-scripting/

d-fine

Less is more –
What do we really need to know from a path to price a derivative?

E.g. (Equity) Spread Option	$V(T) = [S_1(T) - S_2(T)]^+$ underlying **asset values** $S_1(\cdot)$ and $S_2(\cdot)$ at expiry observation time T
E.g. Interest Rate Caplet	$V(T) = \left[L\left(T_{fix}, T_1, T_2\right) - K\right]^+$ with $L\left(T_{fix}, T_1, T_2\right) = \left[\frac{P(T_{fix}, T_1)}{P(T_{fix}, T_2)} D_{12} - 1\right] \frac{1}{T_2 - T_1}$ **zero bonds** $P(\cdot,\cdot)$ for observation time T_{fix} and maturity times T_1, T_2 (1)
Discounting	$V(t) = N(t) \cdot \mathbb{E}[V(\cdot)/N(T)]$ **numeraire** price $N(\cdot)$ at payment observation time T

```
class Path {

  StochProcess* process_;
  MCSimulation* sim_;
  size_t        idx_;

  Path (...) { ... }

  Real asset( Time obsTime,
              string alias ) {
    State* s = sim_->state(idx_, obsTime);
    return process_->asset(obsTime,
                           s, alias);
  }
  real zeroBond( Time t, Time T ) {
    State* s = sim_->state(idx_, t);
    return process_->zeroBond(t, T, s);
  }
  real numeraire( Time obsTime ) {
    State* s = sim_->state(idx_, obsTime);
    return process_->numeraire(obsTime, s);
  }
};
```

The path only knows how to derive a state of the world at observation time and delegates calculation to the underlying stochastic process (or model)

(1) plus deterministic spread discount factor D_{12} to account for tenor basis

d-fine

With the generic path definition the payoff specification becomes very easy

```
class Payoff {
  Time observationTime_;
  virtual Real at(Path* p) = 0;
  virtual Real discountedAt(Path* p) { return at(p) / p->numeraire(observationTime_); }
};
```

```
class Asset : Payoff {

  string alias_;

  virtual Real at(Path* p) {
    return p->asset(
      observationTime_,
      alias_);
  }
};
```

```
class Mult : Payoff {

  Payoff *x_, *y_;

  virtual Real at(Path* p) {
    return
      x_->at(p) * y_->at(p);
  }
};
```

```
class Pay : Payoff {

  Payoff *x_;

  Pay(Payoff *x, Time t)
  : Payoff(t), x_(x) {}

  virtual Real at(Path* p) {
    return x_->at(p);
  }
};
```

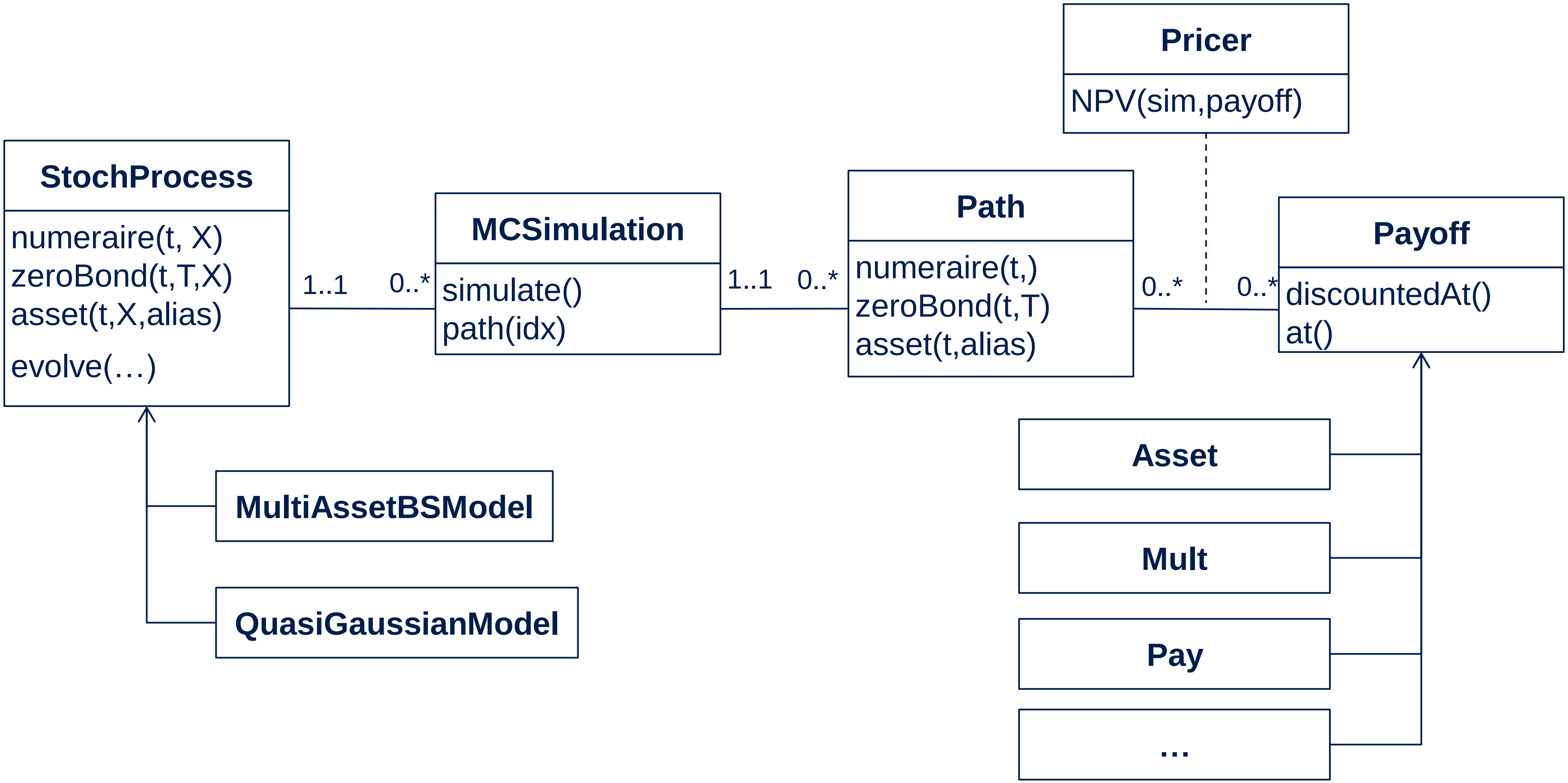
Some consequences

- » The payoff only needs to know a path to calculate its value via `at(.)` method
- » If we want $S(T_1)$ and $S(T_2)$ then we need two payoffs, e.g. `Asset(T1, "S")` and `Asset(T2, "S")`

Once we have a set of elementary payoffs we may combine them to create complex derivative payoffs

d-fine

The big picture…



The chosen architecture allows flexibly addiing new models and payoffs.

d-fine

Another example to illustrate the usage of payoffs…

Today	10.10.2017

YCF-DOM	2.00%	0002f#0001
DIV-S1	3.00%	0002c#0001
DIV-S2	4.00%	0002e#0001

VTSF-S1	20.00%	00033#0001
VTSF-S2	30.00%	0002a#0002

Corr	100%	30%
	30%	100%

Spot-S1 (norm.)	1.00
Spot-S2 (norm.)	1.00

BS-S1	**S1**	00034#0001
BS-S1	**S2**	00036#0002

Model	obj_00037#0005

EndTerm	1y1m
Tenor	1m
Schedule	obj_00030#0001
Npaths	1000
Seed	1
RichEx	FALSE
TimeInterp	TRUE
StoreBrownians	FALSE
MC Simulation	obj_00038#0005
Simulate	TRUE
DoAdjust	TRUE
AssetAdjuster	TRUE

Description	**Payoff-Object**
S1	obj_0003b#0011
S2	obj_0003c#0000
S1 - S2	obj_0003d#0004
0	obj_0003e#0006
[S1 - S2]^+	obj_0003f#0004
Pay	obj_00040#0010

NPV	0.121

Though flexible in principle, assembling the payoff objects manually might be cumbersome.

d-fine

A Flex/Bison-based Parser for a Bespoke Scripting Language

Our scripting language consists of a list of assignments which create/modify a map of payoffs

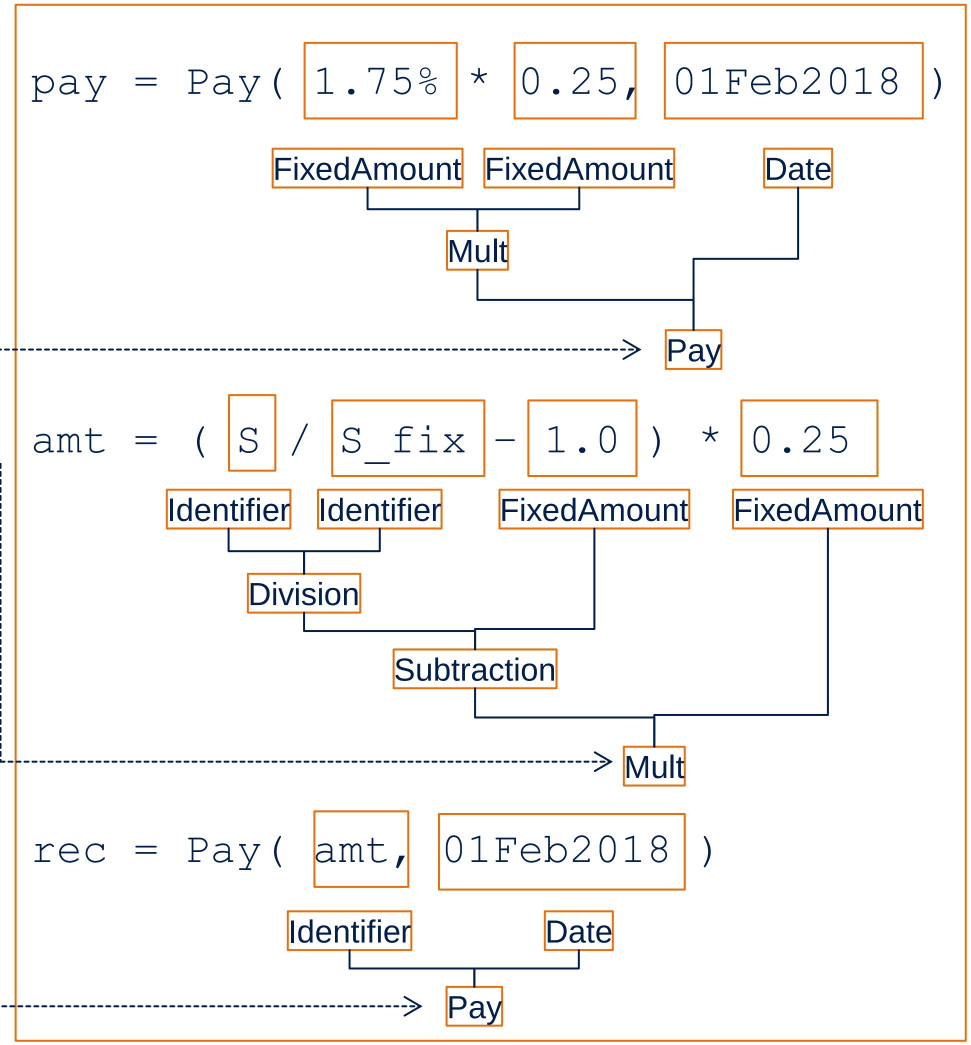
Key	Value
„S_fix“	FixedAmount(100.0)
„S“	Asset(0.25,“SPX“)

```
pay = Pay( 1.75% * 0.25, 01Feb2018 )

amt = ( S / S_fix - 1.0 ) * 0.25

rec = Pay( amt, 01Feb2018 )
```

d-fine

Our scripting language consists of a list of assignments which create/modify a map of payoffs

Key	Value
„S_fix“	FixedAmount(100.0)
„S“	Asset(0.25,“SPX“)
pay	[.]
amt	[.]
rec	[.]



Once the script is parsed the resulting payoffs are accessible via their keys

How do we get from the text input to a QuantLib payoff object?

Scanner

- Define the set of terminal symbols (alphabet, list of tokens) of the language
- Use GNU Flex to generate a scanner for the text input

Parser

- Define the grammar of the scripting language
- Use GNU Bison to generate a parser
  - Utilise the Flex scanner to identify valid tokens in text input
  - Creates an **abstract syntax tree** for a given text input

Interpreter

- Iterate recursively through abstract syntax tree
- Generate QuantLib payoff objects
- Store a reference to final payoff in payoff scripting map

The interface between Scanner/Parser and QuantLib is the abstract syntax tree (AST). In principle, the AST could be generated by other tools as well.

Input scanning is implemented via GNU Flex

- Open source implementation of Lex (standard lexical analyzer on many Unix systems)
- Generates C/C++ source code which provides a function `yylex(.)` which returns the next token

Token definitions

- Operators and punctuations

```
+, -, *, /, ==, !=, <=, >=, <, >, &&, ||, (, ), =, ","
```

- Pre-defined function key-words

```
Pay, Min, Max, IfThenElse, Cache
```

- Identifier

```
[a-zA-Z][a-zA-Z_0-9]*
```

- Decimal number (double)

```
[0-9]*\.?[0-9]+([eE][-+]?[0-9]+)?
```

- Date (poor man’s defintion which needs semantic checking during interpretation phase)

```
[0-9]{2}(Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec)[0-9]{4}
```

Due to automated scanner generation via Flex improvements and extensions are easily incorporated

d-fine

Parse tree generation is implemented via GNU Bison

- Open source implementation of a Lookahead-LR (LALR) parser
- Generates C++ source code with class `Parser` and method `parse(.)` that fascilitates parsing algorithm

Grammar rules (in BNF-style notation)

- A valid string consists of an assignment

```
assignment: IDENTIFIER "=" exp
```

- An expression represents a payoff which may be composed of tokens and other expressions, e.g.

Rule	*Parse Tree*	*Payoff Interpretation*
`exp: exp "+" exp`	create Add-expression	create Add-payoff
`| "(" exp ")"`	pass on expression	pass on payoff in expression
`| IDENTIFIER`	create Identifier-expression	lookup payoff in payoff map
`| NUMBER`	create Number-expression	create fixed amount payoff
`| PAY "(" NUMBER ")"`	create Pay-expression	create Pay-payoff based on year fraction
`| PAY "(" DATE ")"`	create Pay-expression	create Pay-payoff based on date

Due to automated parser generation via Bison improvements and extensions are easily incorporated

d-fine

Payoffs may also be *used* as functions within payoff script

- Derivative payoffs often refere to the same underlying at various dates, e.g.
  - Asset value at various barrier observation dates $S(T_1), \dots, S(T_n)$
  - Libor rate at various fixing dates $L(T_1), \dots, L(T_n)$
- We allow cloning payoffs with modified observation date

Key	Value
„S“	Asset(0.0,“SPX“)

```
amt = ( S( 01Feb2018 ) /
        S( 01Nov2017 ) - 1.0 ) * 0.25

rec = Pay( amt, 01Feb2018 )
```

```
class Asset : Payoff {
  string alias_;

  Asset(Time t, string alias)
  : Payoff(t), alias_(alias){ }

  virtual Asset* at(Time t) {
    return new Asset(t,alias_);
  }
};
```

Eventhough S(.) looks like a function in the script, by means of the parser S(T1) and S(T2) are just two new payoff objects in QuantLib

d-fine

Some Scripting Examples

A „Phoenix Autocall“ Structured Equity Note

Example

- Structured 1y note with conditional quarterly coupons and redemption

Underlying

- Worst-of basket consisting of two assets „S1“ and „S2“
- For briefty initial asset values are normalised to $S_1(0) = S_2(0) = 1.0$

Coupon

- Pay 2% if basket is above 60% at coupon date
- Also pay previous un-paid coupons if basket is above 60% (memory feature)

Autocall

- If basket is above 100% at coupon date terminate the structure
- Pay early redemption amount of 101%

Final Redemption

- If not autocalled pay 100% - DIPut, DIPut with strike at 100% and in-barrier at 60%
- Redemption floored at 30%

A Euribor-linked annuity loan

Example

» Variable maturity loan paying quarterly installments

Installments

» Pay a fixed amount on a quarterly basis

Interest and Redemption Payments

» Interest portion of installment is Libor-3m + 100bp on outstanding notional

» Use remaining installment amount to redeem notional

Maturity

» Loan is matured once notional is fully redeemed

Recursion for Payed Installments and Outstanding Balance

Accrual interest	$Int_i = [L_i + s] \cdot \delta_i \cdot B_i$
Payed installment	$Pay_i = \min\{B_i + Int_i,\ Installment\} = \min\{[1 + (L_i + s) \cdot \delta_i] \cdot B_i,\ Installment\}$
New Balance	$B_{i+1} = B_i - Pay_i$

d-fine

Summary

Summary and Conclusions

Summary

- Flexible payoff scripting requires a clear separation of models, simulations, paths and payoffs
- Payoffs may easily be generated from a small set of interface functions
- Payoff scripting can be efficiently implemented via scanner/parser generators (e.g. Flex/Bison)

Further Features (not discussed but partly implemented already)

- CMS (i.e. swap rate) payoff
- Continuous barrier monitoring
- Regression-based Min-/Max-payoff for American Monte Carlo
- Handling payoffs in the past (with already fixed values)
- Multi-currency hybrid modelling; attaching aliases to ZCB‘s and Euribor payoffs?

> Payoff scripting in QuantLib provides a tool box for lots of fun analysis

d-fine

Dr Sebastian Schlenkrich
Senior Manager
Tel +49 89 7908617-355
Mobile +49 162 2631525
E-Mail Sebastian.Schlenkrich@d-fine.de

Artur Steiner
Partner
Tel +49 89 7908617-288
Mobile +49 151 14819322
E-Mail Artur.Steiner@d-fine.de

d-fine

Frankfurt
München
London
Wien
Zürich

Zentrale

d-fine GmbH
An der Hauptwache 7
D-60313 Frankfurt/Main

Tel +49 69 90737-0
Fax +49 69 90737-200

www.d-fine.com

d-fine