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.
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 \textit{discrete observation} dates
  \[ p = [p_0, \ldots, p_M] \in \mathbb{R}^M \]
- 1-factor model for \textbf{European payoffs}
  \[ p = p_0 \in \mathbb{R} \]

\textbf{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**

```cpp
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/
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

\[ V(T) = \left[ L(T_{fix}, T_1, T_2) - K \right]^+ \]
with
\[ L(T_{fix}, T_1, T_2) = \left[ \frac{P(T_{fix}, T_2)}{P(T_{fix}, T_1)} D_{12} - 1 \right] \cdot \frac{1}{T_2 - T_1} \]
zero bonds \( P(\cdot, \cdot) \) for observation time \( T_{fix} \) and maturity times \( T_1, T_2 \)

Discounting

\[ V(t) = N(t) \cdot \mathbb{E}[V(\cdot)/N(T)] \]
**numeraire** price \( N(\cdot) \) at payment observation time \( T \)

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

---

```cpp
class Path {
    StochProcess* process_;  // Stochastic process or model
    MCSimulation* sim_;      // Monte Carlo simulation
    size_t idx_;             // Index

    Path (...) { ... }  // Path constructor

    Real asset( Time obsTime,  // Asset value
                string alias ) {
        State* s = sim_->state(idx_, obsTime);
        return process_->asset(obsTime, s, alias);
    }

    real zeroBond( Time t, Time T ) {  // Zero bond
        State* s = sim_->state(idx_, t);
        return process_->zeroBond(t, T, s);
    }

    real numeraire( Time obsTime ) {  // Numeraire
        State* s = sim_->state(idx_, obsTime);
        return process_->numeraire(obsTime, s);
    }
};
```
With the generic path definition the payoff specification becomes very easy

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

```c++
class Asset : Payoff {
    string alias;
    virtual Real at(Path* p) { return p->asset(observationTime_, alias_); }
};
```

```c++
class Mult : Payoff {
    Payoff *x_, *y_;  
    virtual Real at(Path* p) { return x_->at(p) * y_->at(p); }
};
```

```c++
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(T_1, "S")` and `Asset(T_2, "S")`

Once we have a set of elementary payoffs we may combine them to create complex derivative payoffs
The chosen architecture allows flexibly adding new models and payoffs.
Another example to illustrate the usage of payoffs...

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

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>„S_fix“</td>
<td>FixedAmount(100.0)</td>
</tr>
<tr>
<td>„S“</td>
<td>Asset(0.25,“SPX“)</td>
</tr>
</tbody>
</table>

\[
\text{pay} = \text{Pay}( 1.75\% \times 0.25, \text{01Feb2018} )
\]

\[
\text{amt} = \left( \frac{S}{S_{\text{fix}}} - 1.0 \right) \times 0.25
\]

\[
\text{rec} = \text{Pay}( \text{amt}, \text{01Feb2018} )
\]
Our scripting language consists of a list of assignments which create/modify a map of payoffs

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;S_fix&quot;</td>
<td>FixedAmount(100.0)</td>
</tr>
<tr>
<td>&quot;S&quot;</td>
<td>Asset(0.25,&quot;SPX&quot;)</td>
</tr>
<tr>
<td>pay</td>
<td>[ . ]</td>
</tr>
<tr>
<td>amt</td>
<td>[ . ]</td>
</tr>
<tr>
<td>rec</td>
<td>[ . ]</td>
</tr>
</tbody>
</table>

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

\[
\text{pay} = \text{Pay}( 1.75\% \times 0.25, \text{01Feb2018} )
\]

\[
\text{amt} = \left( \frac{S}{S_{\text{fix}}} - 1.0 \right) \times 0.25
\]

\[
\text{rec} = \text{Pay}( \text{amt}, \text{01Feb2018} )
\]
### How do we get from the text input to a QuantLib payoff object?

<table>
<thead>
<tr>
<th>Scanner</th>
</tr>
</thead>
<tbody>
<tr>
<td>» Define the set of terminal symbols (alphabet, list of tokens) of the language</td>
</tr>
<tr>
<td>» Use GNU Flex to generate a scanner for the text input</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parser</th>
</tr>
</thead>
<tbody>
<tr>
<td>» Define the grammar of the scripting language</td>
</tr>
<tr>
<td>» Use GNU Bison to generate a parser</td>
</tr>
<tr>
<td>› Utilise the Flex scanner to identify valid tokens in text input</td>
</tr>
<tr>
<td>› Creates an <strong>abstract syntax tree</strong> for a given text input</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Interpreter</th>
</tr>
</thead>
<tbody>
<tr>
<td>» Iterate recursively through abstract syntax tree</td>
</tr>
<tr>
<td>» Generate QuantLib payoff objects</td>
</tr>
<tr>
<td>» Store a reference to final payoff in payoff scripting map</td>
</tr>
</tbody>
</table>

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-Z0-9]*`

- 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
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 facsimulates 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.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Parse Tree</th>
<th>Payoff Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>exp: exp &quot;+&quot; exp</td>
<td>create Add-expression</td>
<td>create Add-payoff</td>
</tr>
<tr>
<td>&quot;(&quot; exp &quot;)&quot;</td>
<td>pass on expression</td>
<td>pass on payoff in expression</td>
</tr>
<tr>
<td>IDENTIFIER</td>
<td>create Identifier-expression</td>
<td>lookup payoff in payoff map</td>
</tr>
<tr>
<td>NUMBER</td>
<td>create Number-expression</td>
<td>create fixed amount payoff</td>
</tr>
<tr>
<td>PAY &quot;(&quot; NUMBER &quot;)&quot;</td>
<td>create Pay-expression</td>
<td>create Pay-payoff based on year fraction</td>
</tr>
<tr>
<td>PAY &quot;(&quot; DATE &quot;)&quot;</td>
<td>create Pay-expression</td>
<td>create Pay-payoff based on date</td>
</tr>
</tbody>
</table>

Due to automated parser generation via Bison improvements and extensions are easily incorporated.
Payoffs may also be *used* as functions within payoff script

» Derivative payoffs often refer to the same underlying at various dates, e.g.
  » Asset value at various barrier observation dates $S(T_1), ..., S(T_n)$
  » Libor rate at various fixing dates $L(T_1), ..., L(T_n)$

» We allow cloning payoffs with modified observation date

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>„S“</td>
<td>Asset(0.0,“SPX“)</td>
</tr>
</tbody>
</table>

```cpp
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 $S(T_1)$ and $S(T_2)$ are just two new payoff objects in QuantLib
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 briefly 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

Accrued interest

\[ \text{Int}_i = [L_i + s] \cdot \delta_i \cdot B_i \]

Payed installment

\[ \text{Pay}_i = \min\{B_i + \text{Int}_i, \text{Installment}\} = \min\{[1 + (L_i + s) \cdot \delta_i] \cdot B_i, \text{Installment}\} \]

New Balance

\[ B_{i+1} = B_i - \text{Pay}_i \]
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
define