Financial Computing with C++

Dmitry Kramkov

Carnegie Mellon University and University of Oxford

2017

Introduction

Elements of syllabus
Results from previous years
Goals of the course

Elements of Syllabus

Materials:
1. Lecture Notes.
2. `cfl` library and the project `Examples`.
3. Books on C++ with **STL** library: Stroustrup, Josuttis, etc..

Requirements:
1. Notebook with C++ compiler.
   - **MS Visual Studio 2017 (Community Desktop Edition)**.
   - **GNU GCC** (5.4.0)
2. "Basic" knowledge of relevant topics:
   - 2.1 Object-oriented programming with C++
   - 2.2 Stochastic Analysis
   - 2.3 Arbitrage-Free Pricing of Derivatives
   - 2.4 Numerical Analysis

Elements of Syllabus

Course work:

	Homeworks	Exam
#	4 (best 3)	4 (best 3)
Collaboration	YES	NO

Extra exercises:

- Problem sessions.
- Sample Exams.
- Bonus homework containing 5 problems of "extra" difficulty.

Results from previous years (MSCF at CMU)

2014: # of students = 42

# of solved problems	4	3	2	1	0
# of students	6	10	11	9	6

2015: # of students = 35

# of solved problems	4	3	2	1	0
# of students	3	11	5	14	2

2016: # of students = 60

# of solved problems	4	3	2	1	0
# of students	5	17	14	13	11

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 (and test!) the ability to use C++ for practical financial computations such as

1. Calibration of financial models, that is, the construction of continuous curves (discount, volatility, forward, etc.) from discrete input data.
2. Pricing of (quite complicated!) real life derivative securities.

Case study: `cfl` library

`cfl` (**Library** for the course **Financial Computing** with C++) is a part of the course package. We shall use it to achieve the goals of the course:

"Theoretical": discuss the design and implementation of a powerful C++ library for pricing of derivative securities.

"Practical": use `cfl` library for practical financial computations.

Remark

The library is intended only for the course. No guarantee is given with respect to the accuracy of the results.

Elements of C++

Bibliography
Namespaces
Inheritance and template
Errors and exceptions
Standard Template Library
Memory management and “smart” pointers
“Pimpl” (pointer to implementation) idiom

Bibliography


Bjarne Stroustrup.
The C++ programming language


Nicolai M Josuttis.
The C++ standard library : a tutorial and handbook.


Scott Meyers.
Effective C++: 55 Specific Ways to Improve Your Programs and Designs.


Mats Henricson, Erik Nyquist.
Industrial strength C++.
(http://hem.passagen.se/erinyq/industrial/).

Namespaces

Examples of namespaces:

- `std`: **STL** (Standard Template Library)
- `boost`: library of "candidates" to STL.
- `cfl`: library for the course "Financial Computing".
- `prb`: namespace for "problems" (examples, homeworks, exams, etc.).

Basic role: group related concepts. The namespaces help to

1. avoid name collisions.
2. organize a large amount of code (such as a library).

Namespaces: name collisions

Assume that we have 4 libraries:

- `std`: **STL** (Standard Template Library)
- `cfl`: library for the course "Financial Computing". `cfl` depends on `std`.
- `finlab`: another library for pricing of derivatives. `finlab` also depends on `std`.
- `prb`: library containing pricing modules for different derivatives. `prb` depends on `std` + `cfl` + `finlab`.

If both `cfl` and `finlab` have the class `Function`, then we have **name conflict**.

Namespaces: names collisions

There are two ways to avoid names conflicts.

1. **Use longer names**: append prefix `cfl` to all classes in `cfl` library:

   $$\texttt{Function} \rightarrow \texttt{cflFunction}.$$

   Disadvantage: leads to "ugly" long names. We get something like `cflGaussRollbackCrankNicolson`, etc.

2. **Use namespaces**: put all classes and functions from `cfl` library into namespace `cfl`.

```
namespace cfl {
  class Function {
    // some code
    };
    }
```

Namespaces: names collisions

To use the class `Function` later we can write

1. everywhere (both in `.hpp` or `.cpp` files):

   ```
   cfl::Function uF;
   ```

2. only in `.cpp` files (never in `.hpp`!):

   ```
   using namespace cfl;
   Function uF;
   ```

Namespaces: code organization

```
namespace cfl { //main namespace
  //global classes:
  class Slice {  ... };
  class IModel { ... };
  class Function { ... };
  //subspaces:
  namespace Black { //Black model
    class Data { ... };
    AssetModel model( ... );
     }
  namespace HullWhite { //Hull and White model
    class Data { ... };
    InterestRateModel model( ... );
     }
  }
```

Namespaces in `cfl` library

cfl:	Main namespace for cfl library.
cfl::Black:	Black model for a single asset.
cfl::Data:	Data structures.
cfl::HullWhite:	Hull and White model for interest rates.
cfl::NApprox:	One-dimensional approximation methods.
cfl::NBrownian:	Implementations of class Brownian.
cfl::NError:	Different types of exceptions.
cfl::NExtended:	Implementations of "expandable" financial models.
cfl::NGaussRollback:	Conditional expectation with respect to gaussian distribution.
cfl::NInd:	Implementations of indicator functions.
cfl::NInterp:	Implementations of different interpolation methods.

Inheritance and templates

Assume that we have to price a *standard call* option.

Black & Scholes model:

$$dS_t = S_t(rdt + \sigma dW_t)$$

Empirical fact: stock price $\uparrow \Leftrightarrow$ volatility $\downarrow$

Elastic model:

$$dS_t = S_t(rdt + \sigma \left(\frac{S_0}{S_t}\right)^{\gamma} dW_t)$$

Here $0 < \gamma < 1$ is an *elasticity* coefficient.

Inheritance and templates

1. Code for Black model:

```
double call(const Black::Model & rModel,
            double dStrike) {
  cfl::Slice uCall =
     max(rModel.spot(1) - dStrike, 0.);
  uCall.rollback(0);
  return atOrigin(uCall);
  }
```

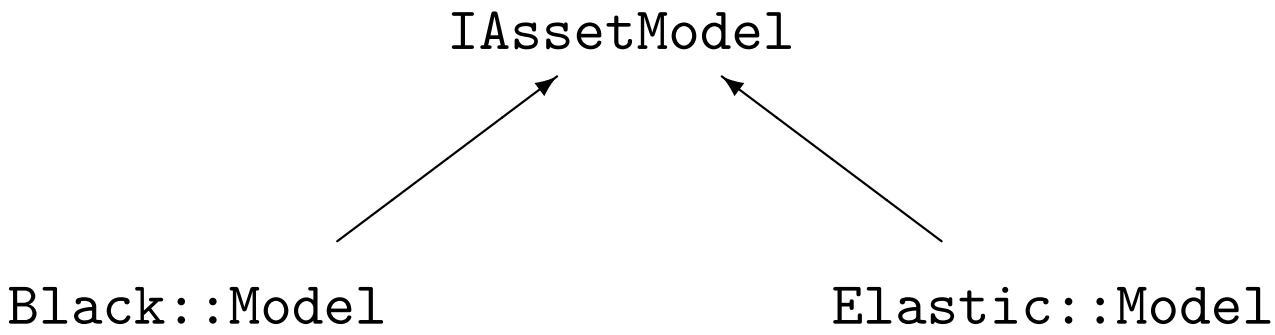
2. Code for Elastic model:

```
double call(const Elastic::Model & rModel,
            double dStrike) {
  // Same code!
  ...
  }
```

Inheritance

Solution 1: use **Inheritance**.

1. Create "abstract" or "interface" class `IAssetModel`. (An abstract class contains only *pure virtual functions*.)
2. Derive `Black::Model` and `Elastic::Model` from `IAssetModel`:



Inheritance

```
class IAssetModel {
  public:
    virtual Slice spot(unsigned) const=0;
  }; //No implementation!
namespace Black {
  class Model: public IAssetModel {
    public:
      Slice spot(unsigned) const;
      ...
    }; //Implement in *.cpp
  }
namespace Elastic {
  //Same code
  ...
  }
```

Inheritance

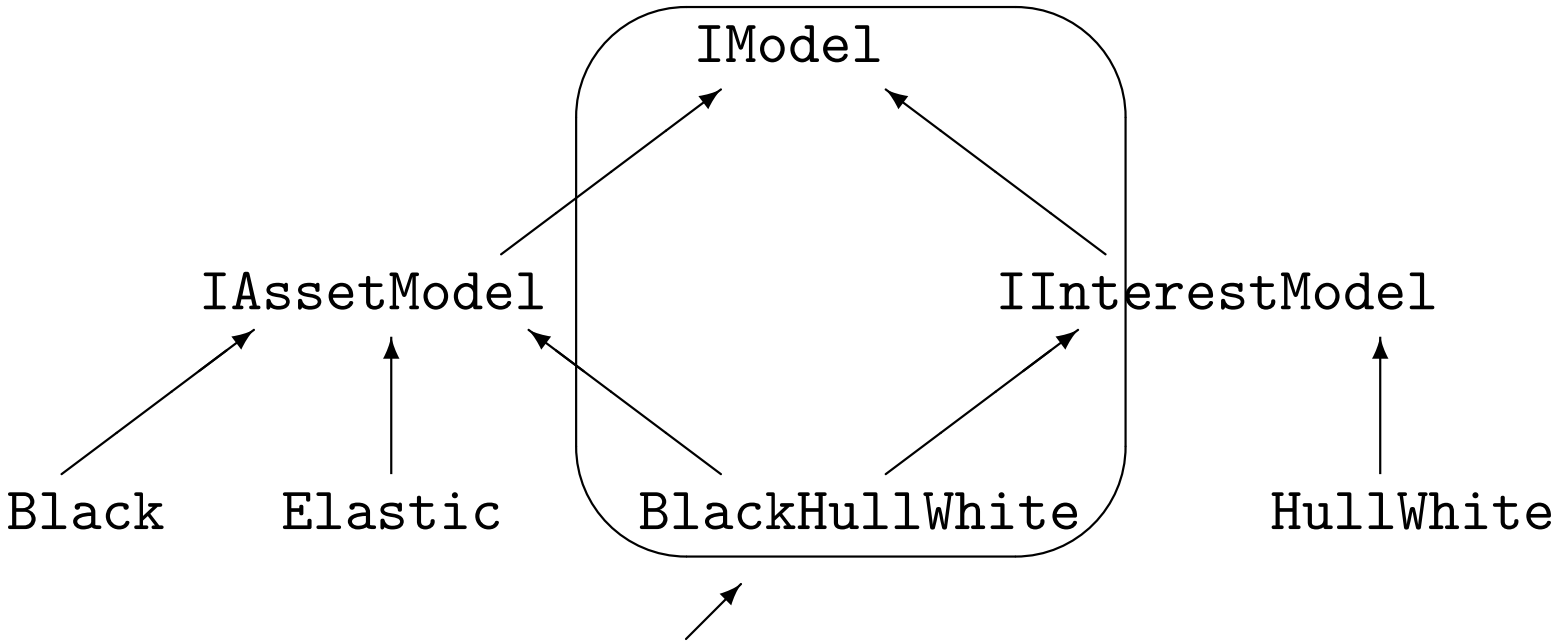
To price a standard call we create the function:

```
double call(const IAssetModel & rModel,
            double dStrike) {
  cfl::Slice uCall =
     max(rModel.spot(1) - dStrike, 0.);
  uCall.rollback(0);
  return atOrigin(uCall);
  }
```

We can use this function for both **Black** and **Elastic** models!

Inheritance: disadvantages

1. Slower execution due to virtual functions.
2. Complicated design:



multiple inheritance $\Rightarrow$ virtual inheritance

Inheritance

Virtual inheritance:

```
class IModel {
  ...
  };
class IAssetModel: public virtual IModel {
  public:
    virtual Slice spot(unsigned) const=0;
    ...
  };
```

Advices for design:

1. Use **abstract** classes.
2. Keep the structure “flat”.

Templates

Solution 2: use **Templates**.

To price a standard call we create the function:

```
template <class Model>
double call(const Model & rModel,
            double dStrike) {
  cfl::Slice uCall =
     max(rModel.spot(1) - dStrike, 0.);
  uCall.rollback(0);
  return atOrigin(uCall);
  }
```

We can use this function for **any** model that has the member function `spot`.

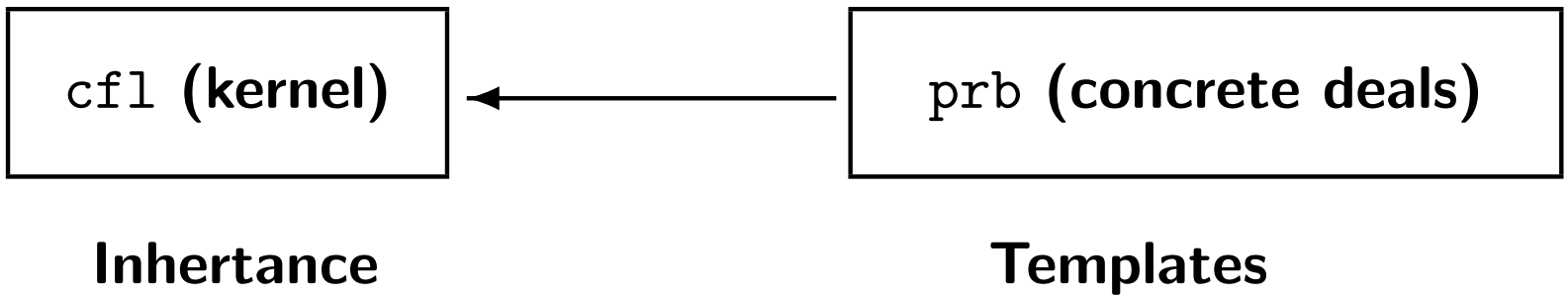
Templates: disadvantages

1. An implementation is “visible” (in `*.hpp` file).
2. A documentation is required.
3. Compilation time is larger.
4. Templates have tendency to propagate throughout the library.

Comparison table

Features	Inheritance	Templates
speed		$\star$
platform independence		$\star$
ease of implementation	$\star$	
documentation	$\star$	
compilation time	$\star$	
visibility of implementation	$\star$	

Possible library design



Remark

`cfl` library is based on the so-called **pimpl** (**pointer to implementation**) design pattern. This software idiom is a flavor of inheritance. We shall discuss it later.

Errors and exceptions

Levels of defense in C++:

1. Compiler
2. Assertions: `assert`.
3. Exceptions: `throw`, `catch` and `try`.

Assertions

The key function: `assert(bool)`.

Example

We want to handle negative volatility in Black and Scholes model.

```
double BSCall(double dSigma, ...) {
  assert(dSigma > 0);
  ...
  }
```

If `dSigma <= 0`, then complier prints the location of the error.
The `assert` command is executed only in `DEBUG` mode.

Assertions

Advantages:
1. Works only in `DEBUG` mode.
2. Documents the code.

Disadvantages: **NONE**, unless you do something complicated or silly.

```
assert(!(iTime == 0)) //good
assert(!(iTime = 0)) //typo!
```

Recommendation: Use `assert` as often as needed.

Exceptions

Key words: `try`, `catch`, `throw`.

When we **write** a function we use `throw`.

```
double BSCall(double dSigma, ...) {
  if(dSigma <= 0) {
    throw(cfl::NError::range(``volatility''))
    }
  ...
  }
```

Exceptions

Later, when we **use** this function, we type:

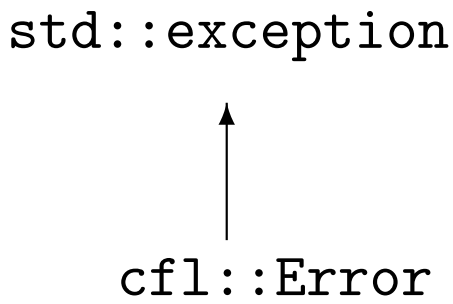
```
try{
  ...
  double dSigma = -1;
  double dCall = BSCall(dSigma, ...);
  ...
  }
catch(const std::exception & rErr) {
  cerr << rErr.what();
  }
```

We should see on the screen:

```
``out of range: volatility''
```

Exceptions in `cfl` library

The basic exception class in `cfl` library is `cfl::Error`. It inherits the basic exception class for STL, namely, `std::exception`:



- A user can continue to `catch` only `std::exception`.
- Helper functions for different types of exceptions, namely, `range`, `sort` and `size`, are collected in namespace `cfl::NError`.

Exceptions

Advantages: 1. work for both `RELEASE` and `DEBUG` mode.
2. allow "grace" reaction to an error: no need to stop the execution of the program.

Disadvantages: *slow down* the performance in the final (`RELEASE`) version.

Recommendation: use with care, mainly, to check external errors.

"Creative" use of exceptions

Goal: perform risk-management analysis of a large portfolio of derivatives (≈ 1000) under various scenarios (≈ 100).

Examples: *VAR (value-at-risk)*, *stress analysis*, etc. .

A typical code:

```
{ ...
  unsigned iN = 100; //number of scenariors
  std::vector<double> uR(iN); //results
  for (i=0; i<iN; i++) {
    //data(i) returns the data for ith scenario
    //value(data(i)) computes the value of the portfolio
    //for input data given by ith scenario
    uR[i] = value(data(i));
    }
  return uR;
}
```

"Creative" use of exceptions

If the data returned by `data(99)` are not valid (for example, they contain negative volatilities), then **all results are lost!**

Use of exceptions:

```
{ ...
  for (i=0; i<iN; i++) {
    try{
      uR[i] = value(data(i));
      }
    catch(...) { //catch all exceptions
      uR[i] = -std::numeric_limits<double>::max();
      }
    }
  return uR;
}
```

STL library

Namespace: `std`.

Classes:

1. `std::vector<T>`
   - 1.1 Elements in memory follow one another.
   - 1.2 Standard container: all STL algorithms work well.
2. `std::valarray<T>`
   - 2.1 Elements in memory follow one another.
   - 2.2 Not a standard container.
   - 2.3 Good for numerical manipulations:

```
uA = uB + uC; // uA[i] = uB[i] + uC[i]
uA = exp(uB); // uA[i] = exp(uB[i])
```

Algorithms: `std::lower_bound`, `std::set_union`, `std::binary_search`, `std::transform`

Memory management

Main rule: # of `new` = # of `delete`.

Example ("Naive" strategy)

```
{
  unsigned iN = numberOfNodes(...);
  //create array of iN numbers
  double * pA = new double [iN];
  ...
  delete [] pA;
}
```

Disadvantages of "naive" strategy

1. Remember (!) to `delete`.
2. Memory leaks with exceptions.

```
{ ...
  try{
    for (i=0; i<iM; i++) {
      unsigned iN = numberOfNodes(...);
      double * pA = new double [iN];
      ... //if error is thrown from here
      //then memory will not be released
      delete [] pA;
      }
    }
  catch(...) {...}
}
```

Efficient memory management

1. Use STL library: no `new` $\Rightarrow$ no `delete`.

```
//instead of
double * pA = new double[iN];
...
delete [] pA;
//use
std::vector<double> uB(iN);
//or
std::valarray<double> uC(iN);
//No problems with exceptions!
```

2. Use "smart" pointers such as `std::auto_ptr<T>` and `std::shared_ptr<T>`.
   (`cfl` library: # of `new` ≈ 100, # of `delete` $= 0$).

std::auto_ptr<T> (depreciated since C++11)

Example

```
class IModel {...}; //interface class
//derived class
class Model: public IModel {...};

// Naive memory management:
IModel * pM1 = new Model(...);
...
delete pM1;

//Use of std::auto_ptr:
std::auto_ptr<IModel> pM2(new Model(...));
...
//no need to delete!
```

std::auto_ptr<T>

Advantages: No need for `delete` command $\Rightarrow$ Safe for exceptions.

Main problem: **terrible** assignment and copy (no `const`!):

```
std::auto_ptr(std::auto_ptr<T> & );
std::auto_ptr & operator= (std::auto_ptr<T> &);
```

1. **Never** create containers of `std::auto_ptr`:

   ```
   std::vector<auto_ptr<T> > uVec; //bad!
   ```

2. Naive code leads to unexpected bad consequences:

   ```
   std::auto_ptr<IModel> pM1(new Model(...));
   std::auto_ptr<IModel> pM2(pM1);
   //pM1 is not valid
   std::auto_ptr<IModel> pM3 = pM2;
   //pM2 is not valid
   ```

std::auto_ptr<T>

Advice: do not use assignment and copy (or be **very** careful).
Replace with `std::unique_ptr` (from C++11).

Example

If `std::auto_ptr` is a private member of a class, then always **implement** or **hide** assignment and copy operators for this class.

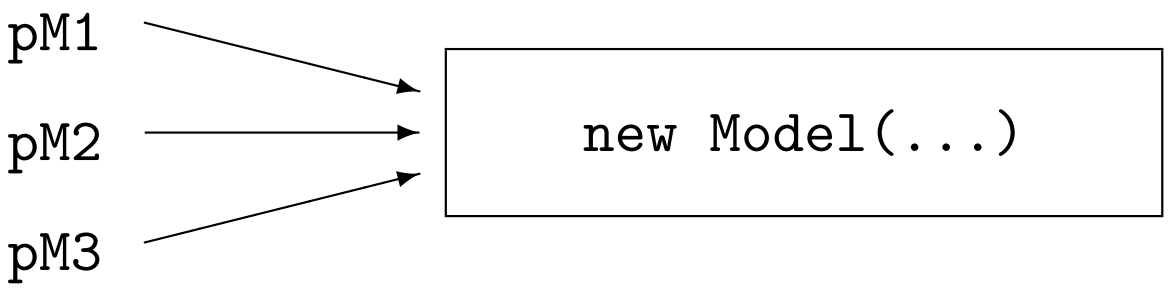
```
class Model
{
  public:
  ...
  private:
    std::auto_ptr<T> m_pT;
    //no need to implement next two
    Model(const Model &);
    Model & operator= (const Model & );
};
```

std::shared_ptr<T> (since C++11)

1. The "smart" pointer for `cfl`.
2. Standard copy and assignment operators (with `const`!):

```
std::shared_ptr(const std::shared_ptr<T> & );
std::shared_ptr & operator =
                    (const std::shared_ptr<T> &);

std::shared_ptr<IModel> pM1(new Model(...));
std::shared_ptr<IModel> pM2(pM1);
std::shared_ptr<IModel> pM3=pM2;
//pM1, pM2 and pM3 point to the same object
```



std::shared_ptr<T>

Advantages: copy and assignment operators are very fast (**"shallow"' copies**). Great for STL containers!

Problem: delicate behavior for classes with **non const** member functions.

```
{
...
pM1->changeData(rNewData);
//pM2 and pM3 point now to model with
//different input data
...
}
```

Advice: use only for classes, where all member functions are `const`.

Pimpl idiom

Main idea: "hide implementation".

Case study: implementation of a one-dimensional function object.
Examples:

1. Input curves for financial models: discount, volatility, forward, etc.
2. Output for prices of derivatives, etc.

Techniques:
1. Templates
2. (Naive) inheritance
3. **Pimpl** (pointer to implementation) pattern.

Template

If we base our design on templates (like in STL), then a function is **any** object with member function similar to:

```
double operator()(double) const;
```

Remark

Nice, but then we get templates everywhere!

Naive inheritance

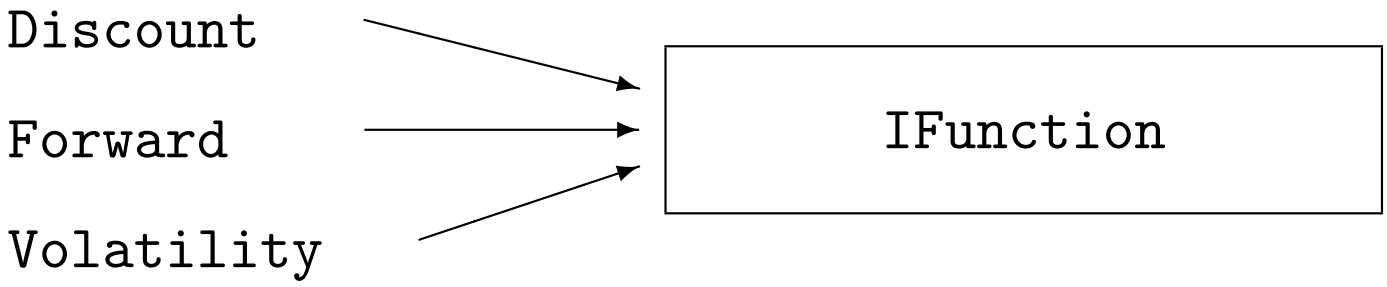
If we base our design on "naive" inheritance, then we

1. define interface class `IFunction`:

```
class IFunction {
  public:
    virtual double
      operator()(double) const = 0;
  };
```

2. call a function any class that implements `IFunction`.

Naive inheritance



Disadvantages: 1. Too many classes will appear.

2. It is not easy to define elegant arithmetic operators.

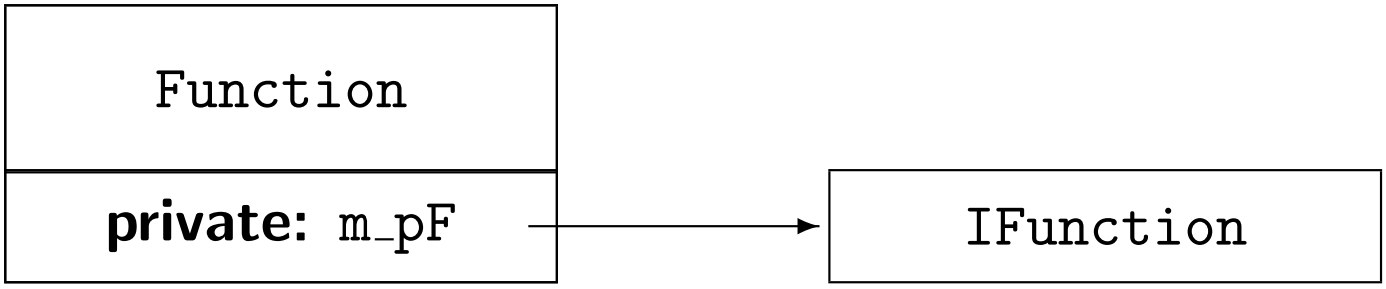
Pimpl idiom

If we base our design on "pimpl" (pointer to implementation) pattern, then we

1. define interface class `IFunction` as before.
2. define the "uniform" class for all functions:

```
class Function {
  public:
    Function(IFunction * pF)
      :m_pF(pF) {}
    ...
    double operator()(double dX) const {
      return (*m_pF)(dX);
      }
  private:
    std::shared_ptr<IFunction> m_pF;
};
```

Pimpl idiom



Advantages:
1. All functions share the same name `Function`.
2. Implementations are done in `*.cpp` files. (We are *hiding* implementations!).
3. It is easy to define and implement elegant and efficient numerical operators such as $+$, $-$, $*$, $/$, exp, max, etc..

Construction of data curves

Function objects in cfl
Interpolation
Least square fitting
Examples

Function objects in `cfl`

`cfl` contains the following function objects:

1. One-dimensional (`cfl::Function`, `cfl:IFunction`)
2. Multi-dimensional (`cfl::MultiFunction`, `cfl:IMultiFunction`)

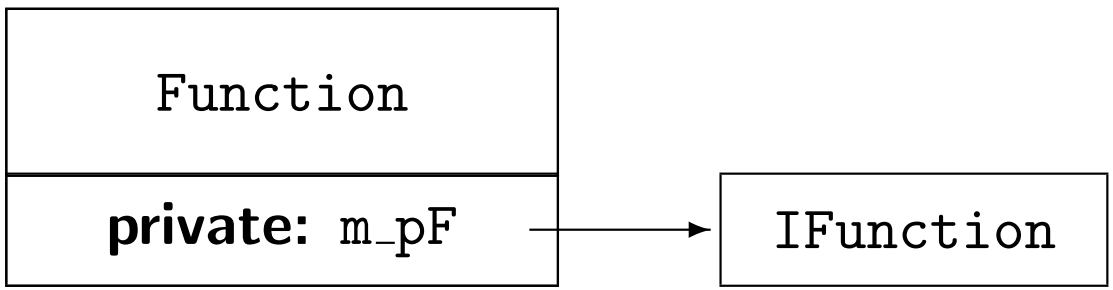
These classes play very important role. For example, we use them to define

- Input curves (volatility, forward, discount, etc).
- Output curves for the prices of derivatives (dependence of computed prices on model's factors).

cfl::Function

Description: the class in `cfl` for a one-dimensional function object.

Implementation: a dynamically allocated object derived from the interface `cfl::IFunction` (**pimpl** pattern).

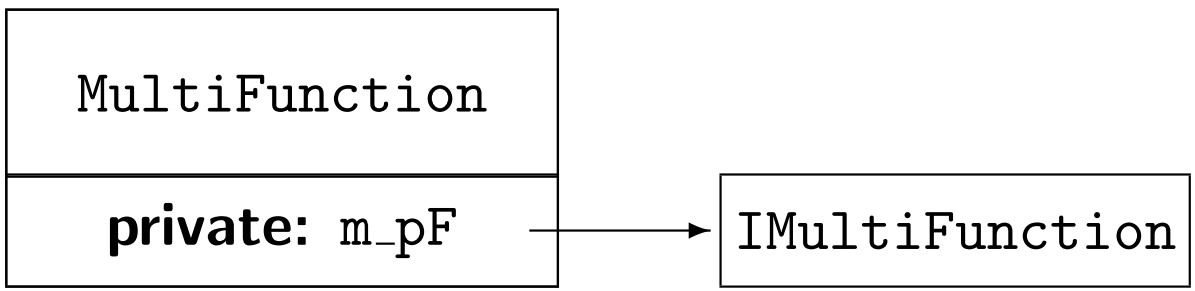


Private members: `std::shared_ptr<IFunction> m_pF`
(all member functions in `IFunction` are `const`!)

cfl::MultiFunction

Description: the class in `cfl` for a multi-dimensional function object.

Implementation: a dynamically allocated object derived from the interface `cfl::IMultiFunction` (**pimpl** pattern).



Private members: `std::shared_ptr<IMultiFunction> m_pF`
(all member functions in `IMultiFunction` are `const`!)

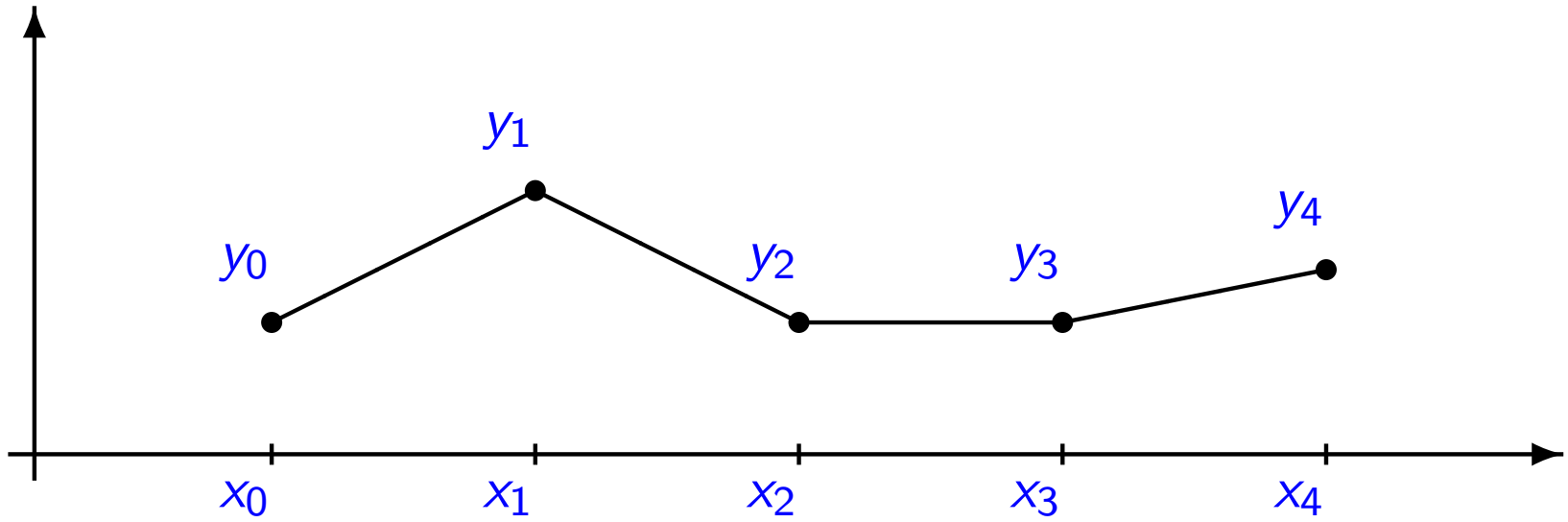
Interpolation

Basic idea: given

$(x_0, \ldots, x_n)$: arguments

$(y_0, \ldots, y_n)$: values of function $f = f(x)$ $(y_k = f(x_k))$

$\Rightarrow$ **restore** $f = f(x)$ for all x.



Popular methods: 1. linear interpolation

2. cubic spline interpolation

Cubic spline

Given the arrays of arguments $(x_k)_{0 \le k \le n}$ and values $(y_k)_{0 \le k \le n}$ the cubic spline is defined as a function $f = f(x)$ such that

1. $f(x_k) = y_k$ for all pairs (x_k, y_k)
2. $f = f(x)$ is a cubic polynomial on every $[x_k, x_{k+1}]$ and is *twice* continuously differentiable (that is, $f'' = f''(x)$ is continuous)
3. Boundary conditions:
   - 3.1 Natural spline: $f''(x_0) = f''(x_n) = 0$.
   - 3.2 Fixed values for derivatives at end points: $f'(x_0) = a$ and $f'(x_n) = b$ for some a and b.

Cubic spline

Minimum norm characterization: among all interpolating functions that satisfy one of the boundary conditions of item 3 the corresponding cubic spline minimizes the $\mathbf{L}^2$ norm for its second derivative:

$$\int_{x_0}^{x_n} (f''(t))^2 dt \to \min .$$

Intuitively, cubic spline is the smooth (with continuous second derivative) interpolating function which has the *least* deviation from the linear interpolating function.

Interpolation in `cfl` library

Numerical interpolation is implemented in `cfl` through the following classes:

1. the interface class `cfl::IInterp`,
2. the concrete class `cfl::Interp`.

These two classes interact with each other by *pimpl* idiom. Concrete implementations are collected in the namespace `cfl::NInterp`.

Least square fitting

Inputs: 1. arguments $(x_1, \ldots, x_n)$
2. values $(y_1, \ldots, y_n)$
3. weights $(w_1, \ldots, w_n)$, $w_i > 0$.
4. a parametric family of functions $f(\lambda) = f(\lambda, x)$, where $(\lambda = (\lambda_1, \ldots, \lambda_m))$ is a parameter.

Output: function $f(\lambda^*)$, where the optimal parameter λ^* is the solution of the least square minimization problem:

$$\sum_{i=1}^{n} w_i (f(\lambda, x_i) - y_i)^2 \to \min .$$

Fast numerical methods are available for *linear least square fitting*, where

$$f(\lambda, x) = \sum_{j=1}^{d} \lambda_j g_j(x)$$

Examples of data curves

The following data curves are constructed in the project Examples.

1. Shape curve for yield changes in Hull and White model (`Examples/Src/DataYieldShapeHullWhite.cpp`).
2. Discount curve obtained by log linear interpolation. (`Examples/Src/DataDiscountLogLinearInterp.cpp`).
3. Discount curve for Hull and White model obtained by least square fitting of market yields. (`Examples/Src/DataDiscountFitHullWhite.cpp`).

These functions are declared in `Examples/Examples.hpp`.

Abitrage-free pricing theory

Classification of derivative securities
Rollback operator
State processes
Model implementation

Classification of derivatives

Type of underlying: (important for design!)

1. Assets: stocks, FX rates, commodities, etc.
2. Interest rates.

Dependence on the history: (important for numerical implementation!)

1. Standard: payoff depends on the current market values
2. Path dependent: payoff depends on historical values
3. Barrier: simple dependence on the past

Example: American butterfly

This is an example of **Standard Asset** option.

P : strike for put option

C : strike for call option ($C > P$)

$(t_i)_{1 \leq i \leq N}$: exercise times

At an exercise time t_i a holder of the option can

1. **sell** the underlying stock for the strike P of put option
2. **buy** the underlying stock for the strike C of call option
3. **do nothing**, wait and exercise later.

Example: American Swaption

This is an example of **Standard Interest Rate** option.

$(t_i)_{1 \le i \le n}$: exercise times

Underlying Swap with given parameters:

- notional amount,
- fixed swap rate,
- time interval between two payments (as year fractions),
- total number of payments,
- side of the contract (pay "fixed" and receive "float" or otherwise).

A holder of the option can enter into the underlying swap agreement at any exercise time t_i. This time then becomes the issue time for the swap.

Example: down-and-out Call

This is an example of **Barrier Asset** derivative security.

L : lower barrier

$(t_i)_{1 \leq i \leq M}$: barrier times

K : strike

T : maturity ($T > t_M$).

The payoff of the option at maturity equals

1. the payoff of the standard call option if the spot price was above the barrier for **all** barrier times t_j;
2. zero if the barrier was crossed at a barrier time.

Example: Convertible Bond

This is an example of **Path Dependent Asset** option.

Coupon bond with standard parameters: face value, coupon rate, maturity, etc.

Exercise option for holder: right to sell the bond for a number of shares determined by the strike. The strike is **resettable** (path dependence!).

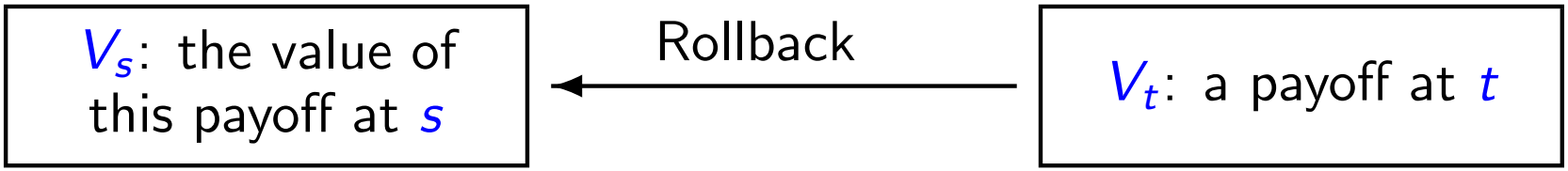
Put option for holder: right to sell for a fixed redemption amount (usually smaller than the face value).

Hard call option for issuer: right to buy for a fixed redemption amount (usually larger than the face value).

Soft call option for issuer: right to buy for a fixed redemption amount if the price of stock exceeds some barrier.

Put option for issuer at maturity: right to sell the bond at maturity for a number of shares determined by a **mandatory strike**.

Rollback operator



Notation:

$$V_s = \mathcal{R}_s[V_t]$$

Main principle:

Arbitrage-Free Pricing = Replication

Convenient method: (for complete financial models)

Arbitrage-Free Pricing = Risk-neutral valuation

Money market measure

Denote

r_t: short-term interest rate at t.

$B_t = \exp(\int_0^t r_u du)$: bank account

The **money market** (martingale) measure $\mathbb{P}^*$ is defined as such a measure that

$$\frac{X_t}{B_t} = X_t \exp(-\int_0^t r_u du)$$

is a **martingale** under $\mathbb{P}^*$ for any wealth process X:

$$X_s \exp(-\int_0^s r_u du) = \mathbb{E}_s^*[X_t \exp(-\int_0^t r_u du)]$$

$$\Updownarrow$$

$$X_s = \mathbb{E}_s^*[X_t \exp(-\int_s^t r_u du)]$$

Risk-neutral valuation

Theorem

The rollback operator has the following representation in terms of the money market measure $\mathbb{P}^*$*:*

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

Proof.

Definition of $\mathbb{P}^*$ + "Arbitrage-Free Pricing = Replication". □

Remark

$$\text{Computation of } \mathcal{R}_s[.] \Leftrightarrow \text{ Computation of } \mathbb{E}_s^*[.]$$

Forward measure

$B(s,t)$: price at s of the zero-coupon bond with face value \$1 and maturity t.
The **forward** (martingale) measure $\mathbb{P}^t$ is such a measure that

$$\frac{X_s}{B(s,t)}, \quad 0 \leq s \leq t,$$

is a **martingale** under $\mathbb{P}^t$ for any wealth process X:

$$\frac{X_s}{B(s,t)} = \mathbb{E}_s^t[X_t]$$

$$\Updownarrow$$

$$X_s = B(s,t)\mathbb{E}_s^t[X_t]$$

Forward measure

The term **forward martingale measure** is due to the fact, that $(F(s,t))_{0\le s\le t}$ is $\mathbb{P}^t$-martingale, where

$F(s,t)$: forward price computed at s for delivery at t.

Indeed, consider long position in the forward contract:

$X_s = 0$: value at s

$X_t = S_t - F(s,t)$: value at t

Then

$$0 = X_s = B(s,t)\mathbb{E}_s^t[X_t] = B(s,t)\mathbb{E}_s^t[S_t - F(s,t)]$$

and, hence,

$$F(s,t) = \mathbb{E}_s^t[S_t]$$

Risk-neutral valuation

Theorem

The rollback operator has the following representation in terms of the forward martingale measure $\mathbb{P}^t$:

$$V_s = \mathcal{R}_s[V_t] = B(s,t)\mathbb{E}_s^t[V_t]$$

Proof.

Definition of $\mathbb{P}^t$ + "Arbitrage-Free Pricing = Replication". □

Remark

To implement $\mathcal{R}_s[V_t]$ we need to implement $\mathbb{E}_s^t[V_t]$.

Problem on float rate

Problem

Notations:

$L(s,t)$: *float (LIBOR) rate computed at time* s *for maturity* t

$B(s,t)$: *discount factor computed at time* s *for maturity* t.

Compute (express in terms of $B(s,t)$*) the value at time* s *of the float payment at* t*. Recall that*

$$\text{“Float payment at } t\text{”} = L(s,t)(t-s).$$

Problem on foreign exchange rates

Problem

Notations:

$F(s,t)$: *forward exchange rate (price of one unit of foreign currency) computed at time* s *for maturity* t

$B(s,t)$: *discount factor (in domestic currency) computed at time* s *for maturity* t.

Compute (express in terms of $F(s,t)$ *and* $B(s,t)$*) the value at time* s *of one unit of foreign currency paid at* t.

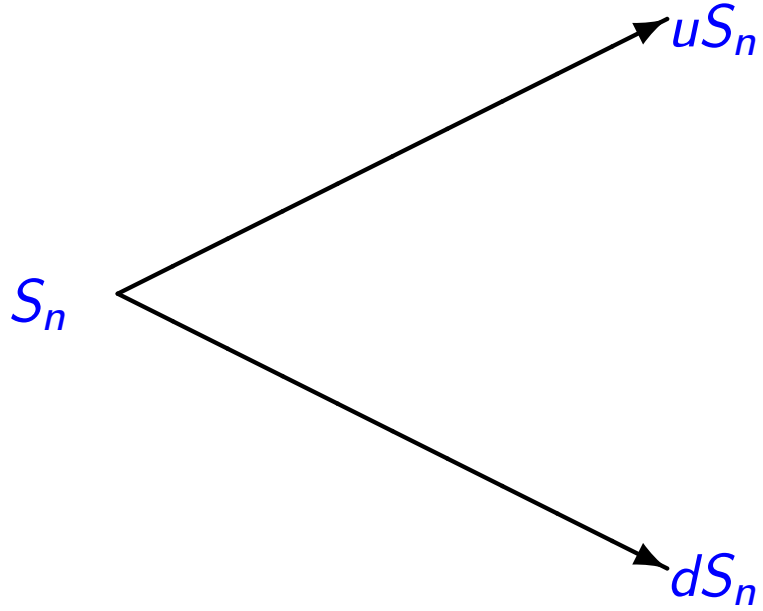
State processes

Idea: efficient storage for relevant random variables.

Consider, for example, **binomial model** with parameters:

u: relative change "up"

d: relative change "down"



Binomial model

Consider payment at $n+1$:

$$V_{n+1} = V_{n+1}(\omega_1, \ldots, \omega_{n+1})$$

Rollback operator between $n+1$ and n has the form:

$$\begin{aligned} V_n &= \mathcal{R}_n[V_{n+1}](\omega_1, \ldots, \omega_n) \\ &= \frac{1}{1+r}[\widetilde{p} V_{n+1}(\omega_{n+1} = H) + \widetilde{q} V_{n+1}(\omega_{n+1} = T)] \end{aligned}$$

where $\widetilde{p}$ and $\widetilde{q}$ are the one-step risk neutral probabilities:

$$\widetilde{p} = \frac{1+r-d}{u-d}, \quad \widetilde{q} = 1 - \widetilde{p}.$$

"Naive" storage

"Naive" storage scheme: record values of

$$V_n = V_n(\omega_1, \ldots, \omega_n)$$

for **all** $(\omega_1, \ldots, \omega_n)$. Then

$$\#\text{ of records } = 2^n \text{ (too big!)}$$

The naive (universal) storage scheme is not practical for already $n \approx 100$.

Idea: the storage should be *adapted* to the *type* of derivative security we want to evaluate.

Storage for standard options

For example, to price *standard options* it is sufficient to operate with random variables in the form

$$V_n = f(S_n),$$

where $f = f(x)$ is a deterministic function. In this case,

$$\#\text{ of records } = n + 1 \text{ (fine!)}$$

Indeed, if $V_{n+1} = f_{n+1}(S_{n+1})$ then

$$V_n = \mathcal{R}_n[V_{n+1}] = f_n(S_n)$$

where $f_n(x) = \frac{1}{1+r}[\widetilde{p} f_{n+1}(ux) + \widetilde{q} f_{n+1}(dx)]$.

State processes

The spot price process $(S_n)_{0\le n\le N}$ in binomial model is an example of a *state process*.

Definition

A process $(X_t)_{0\le t\le T}$ is called a **state process** if $\forall s < t$ 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 processes

Remark

For a stochastic process $X = (X_t)_{0 \le t \le T}$ and time t denote by

$$\mathcal{X}_t = \{ f(X_t) : \quad f \text{ is deterministic function } \}$$

the family of random variables determined by (measurable with respect to) X_t. We have that

For arbitrary X: for any time t the family $\mathcal{X}_t$ is *closed* under any arithmetic or functional operation

For state process X: for two times $s < t$ the families $\mathcal{X}_s$ and $\mathcal{X}_t$ are *closed* under the rollback operator $s \leftarrow t$: for any $V_t \in \mathcal{X}_t$

$$V_s = \mathcal{R}_s[V_t] \in \mathcal{X}_s$$

Markov processes

Definition

A stochastic process $X = (X_t)_{0 \le t \le T}$ is called a **Markov process** if for any $s < t$ and any deterministic $f = f(x)$ there is a deterministic $g = g(x)$ such that

$$g(X_s) = \mathbb{E}_s[f(X_t)]$$

Remark (Intuitive definition)

At time t the future behavior of X (the distribution of $(X_s)_{t \le s}$) is completely determined by the *current value* X_t (does not depend on the particular trajectory between times 0 and t).

State and Markov processes

Theorem

The following conditions are equivalent:

1. X *is a state process*
2. *for any maturity* T
   - 2.1 $(X_t)_{0 \le t \le T}$ *is a Markov process under the forward measure* $\mathbb{P}^T$
   - 2.2 *the discount factor computed at* t *for maturity* T *is determined by* X_t*, that is,*

$$B(t,T) = f(X_t)$$

for some deterministic $f = f(x)$

Proof.

Formula for rollback operator:

$$\mathcal{R}_t(V_T) = B(t,T)\mathbb{E}_t^T[V_T].$$

"Implementation" of a financial model

An "implementation" of a financial model consists of

1. Specification of a state process X (the choice of state process is determined by the type of derivative security).
2. Implementation of all necessary operations for random variables from the classes

$$\mathcal{X}_t = \{f(X_t) : \ f \text{ is a deterministic function } \}, \ t > 0$$

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

Design of `cfl`

Basic elements of `cfl`
Main class cfl::Slice
Design of `cfl`

Basic components

State process:

$$X = (\underbrace{X^0, \ldots, X^{d-1}}_{d-\text{dimensional}})$$

Vector of event times: (initial time =) $t_0 < t_1 < \cdots < t_N$

At an event time t_i we operate with random variables:

$$\mathcal{X}_{t_i} = \left\{ f(X_{t_i}) = f(X^0_{t_i}, \ldots, X^{d-1}_{t_i}) : \quad f = f(x^0, \ldots, x^{d-1}) \right\}.$$

A random variable $f(X_{t_i})$ is represented by the class `cfl::Slice`.

Event times

Any model in `cfl` has *discrete* time structure:

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

t_0: *initial time*.

Event times: all times needed to price a *particular* derivative security. Examples:

1. Exercise times
2. Maturity
3. Barrier times
4. Reset times

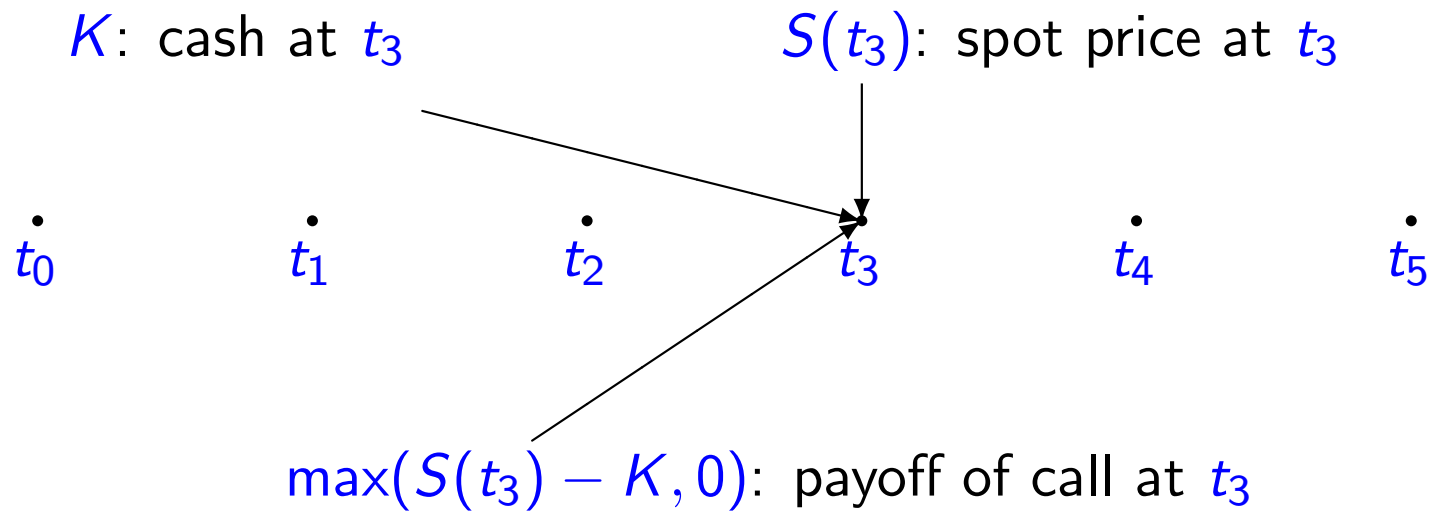
Numerical efficiency: create the vector of event times with a *smallest* size.

cfl::Slice

The main class in the library is `cfl::Slice`.

Basic idea: `cfl::Slice` represents the value of a (derivative) security at a **particular** event time.

Precise defintion: `cfl::Slice` describes random variables in the form $f(X(t_i))$, where $f = f(x)$ is a deterministic function, X is a state process and t_i is an event time.



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 the 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

However, if

$t_1 < t_2$: event times

uSpot1: cfl::Slice for the spot price $S(t_1)$ at t_1

uSpot2: cfl::Slice for the spot price $S(t_2)$ at t_2

then the following code

```
Slice uSum = uSpot1 + uSpot2
```

is **wrong** (should not compile)!

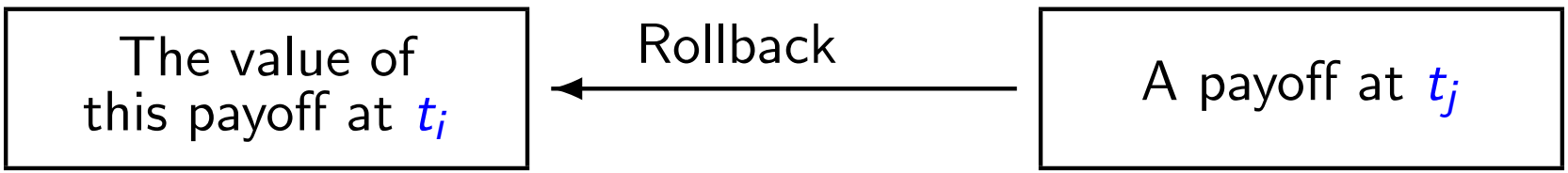
Remark

You might want to create a cfl::Slice for $S(t_1) + S(t_2)$, (say, to price an Asian option), but this operation is not allowed!

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:

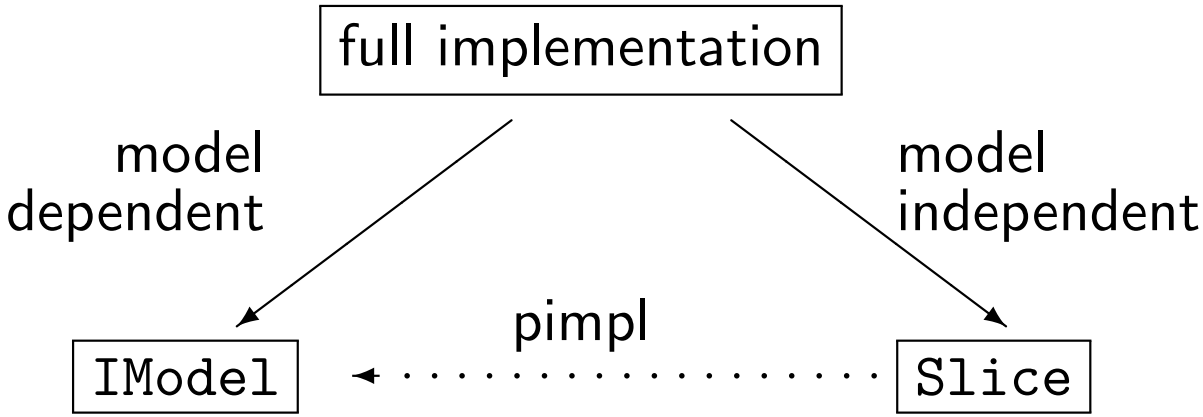
```
...
//two event times:  0 (initial) and 1 (maturity)
Slice uCall = max(uModel.spot(1) - dK, 0);
uCall.rollback(0);
```

Typical 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.

Architecture of `cfl`



abstract (interface) class that defines model-specific behavior of the class `Slice`

concrete class for random functions $f(X_{t_k})$

f: deterministic function

t_k: event time

X: state process

cfl::Slice and cfl::IModel

The class `Slice` represents the random variable in the form: $f(X_{t_k}^{i_1}, \ldots, X_{t_k}^{i_m})$.
Components (private members) of `Slice`:

1. array of values (`std::valarray<double>`): discretization of $f = f(x^{i_1}, \ldots, x^{i_m})$
2. vector of dependences: $i_1, \ldots, i_m$
3. (index of) current event time t_k
4. **pimpl** of `IModel`

The interface class `IModel` contains declarations of **model-specific** functions needed to define the behavior of the class `Slice`.

Single asset models

"Standard" Black model

"Generalized" Black model

Black model in `cfl`

Class cfl::AssetModel

Examples of derivatives

Black and Scholes model

Classical Black and Scholes:

$$dS_t = S_t(rdt + \sigma dW_t),$$

where

S_t: the stock price at t

r: the short-term interest rate

σ: the volatility

W: the Brownian motion under the risk-neutral probability.

For *calibration* purposes we often use *time-dependent* version:

$$r \to (r_t)_{t\geq 0}, \quad \sigma \to (\sigma_t)_{t\geq 0}.$$

Black and Scholes model

Black and Scholes model with dividends:

$$dS_t = S_t((r_t - d_t)dt + \sigma_t dW_t),$$

where d_t is the dividend rate at time t

Black and Scholes model with cost of carry:

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

where q_t is the cost of carry rate at t.

Black and Scholes fx model:

$$dS_t = S_t((r_t^d - r_t^f)dt + \sigma_t dW_t)$$

where r_t^d and r_t^f are domestic and foreign short-term rates at t.

Standard Black model

Key idea: write the model in terms of forward prices. Denote by $F(t, T)$ forward price at t for contract with maturity T.

Black model:

$$dF(t, T) = F(t, T)\sigma_t dW_t$$

or, equivalently,

$$F(t, T) = F(0, T)\exp\left(\int_0^t \sigma_u dW_u - \frac{1}{2}\int_0^t \sigma_u^2 du\right)$$

Input parameters for Black model

$(F(0,T))_{T\geq 0}$: initial forward curve; $F(0,T)$ denotes the forward price (at initial time 0) in the contract with maturity T.

$(B(0,T))_{T\geq 0}$: initial discount curve; $B(0,T)$ is the price of zero-coupon bond with face value $\$1$ and maturity T

$(\Sigma(0,T))_{T\geq 0}$: the *average* historical volatility curve for the spot price on the interval $[0,T]$:

$$\Sigma(0,T)=\sqrt{\frac{1}{T}\int_0^T \sigma^2(u)du},$$

Equivalently, $\Sigma(0,T)$ is the implied volatility of standard European options with maturity T.

Examples of forward curves

1. Classical Black and Scholes without dividends:

$$F(0,T) = S(0)/B(0,T) = S(0)\exp(\int_0^T r_u du)$$

2. Black and Scholes model with cost of carry $(q_t)_{t\geq 0}$:

$$F(0,T) = S(0)e^{\int_0^T q(u)du}$$

3. FX model:

$$F(0,T) = S(0)\exp(\int_0^T (r_u^d - r_u^f)du)$$

Movements of forward prices

From the model evolution:

$$dF(t,T) = F(t,T)\sigma_t dW_t$$

we deduce that forward prices move identically (in relative terms):

$$S(t) \uparrow 1 \text{ b.p} \Leftrightarrow F(t,T) \uparrow 1 \text{ b.p}$$

or, equivalently,

$$\frac{\Delta S(t)}{S(t)} = 0.01\% \Leftrightarrow \frac{\Delta F(t,T)}{F(t,T)} = 0.01\%$$

for any maturity T.

Stationary implied volatility

We also deduce that the implied volatility:

$$\Sigma(t,T) = \sqrt{\frac{1}{T-t}\int_t^T \sigma^2(u)du},$$

is **stationary**:

$$\Sigma(0,T) = \Sigma(t,T+t) \quad t>0, T>0,$$

if and only if

$$\Sigma(t,T) = \sigma(= \text{const})$$

Empirical facts about oil prices

1. Forward prices with longer maturities move **slower**:

$$\frac{\Delta S(t)}{S(t)} > \frac{\Delta F(t,T)}{F(t,T)} > \frac{\Delta F(t,U)}{F(t,U)}, \quad t < T < U.$$

2. Shorter maturities have **larger** implied volatilities:

$$\sigma_t > \Sigma(t,T) > \Sigma(t,U), \quad t < T < U.$$

"General" Black model

The evolution of forward prices:

$$dF(t,T) = F(t,T)A(T)\sigma_t dW_t$$

or, equivalently,

$$F(t,T) = F(0,T)\exp\left(A(T)\int_0^t \sigma_u dW_u - \frac{1}{2}A^2(T)\int_0^t \sigma_u^2 du\right)$$

Here

$(A(T))_{T\geq 0}$: initial "shape curve" for the relative changes in forward prices. It is convenient to assume that

$$A(0) = 1.$$

Movements of forward prices

From the model evolution:

$$dF(t,T) = F(t,T)A(T)\sigma_t dW_t$$

we deduce that:

$$S(t) \uparrow A(t) \text{ b.p} \Leftrightarrow F(t,T) \uparrow A(T) \text{ b.p}$$

or, equivalently,

$$\frac{\Delta S(t)}{S(t)} = A(t)\% \Leftrightarrow \frac{\Delta F(t,T)}{F(t,T)} = A(T)\%.$$

Implied volatilities

The implied volatility for maturity t observed at s:

$$\Psi(s,t) = A(t)\sqrt{\frac{1}{t-s}\int_s^t \sigma^2(u)du}$$

The implied volatility is stationary, i.e. $\Psi(s,t) = \Psi(0,t-s)$ if there are constants $\lambda \geq 0$ and $\kappa > 0$:

$$\begin{aligned} A(t) &= \exp(-\lambda t) \\ \sigma(t) &= \kappa \exp(\lambda t) \\ \Psi(s,t) &= \kappa\sqrt{\frac{1-\exp(-2\lambda(t-s))}{2\lambda(t-s)}} \end{aligned}$$

"Traditional" form

Often the model is written in terms of the spot prices:

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

Here $(\theta)_{t \geq 0}$ is chosen to calibrate input forward curve and $(\lambda_t)_{t \geq 0}$ is the mean-reversion rate

$$\lambda_t = -\frac{A'(t)}{A(t)} \geq 0,$$

or, equivalently,

$$A(t) = \exp\left(- \int_0^t \lambda_u du \right).$$

Inputs for generalized Black model

$(F(0,T))_{T\geq 0}$: initial forward curve;

$(B(0,T))_{T\geq 0}$: initial discount curve;

$(A(T))_{T\geq 0}$: initial "shape curve" for the relative changes in forward prices; $A(0)=1$.

$(\Sigma(0,T)))_{T\geq 0}$: the *normalized* implied volatility curve

$$\Sigma(0,T)=\sqrt{\frac{1}{T}\int_0^T \sigma^2(u)du}=\frac{\Psi(0,T)}{A(T)}.$$

Black model in `cfl`

Generalized (and standard) Black model is implemented in the namespace `cfl::Black`:

1. The class `cfl::Black::Data` defines the parameters of the Black model. Recall, that the set of parameters consists of
   - initial time (as year fraction),
   - discount curve,
   - forward curve,
   - volatility curve,
   - "shape" curve.
2. The functions `cfl::Black::model` implement the *"standard"* single asset model in the library, namely, the class `cfl::AssetModel`.

Output for Black model

It is important (for example, for risk-management) to compute the value of an option in Black model as the function of **relative change** in the price of the stock:

$$V = (V(x))_{\frac{\Delta}{2} \leq x \leq \frac{\Delta}{2}}$$

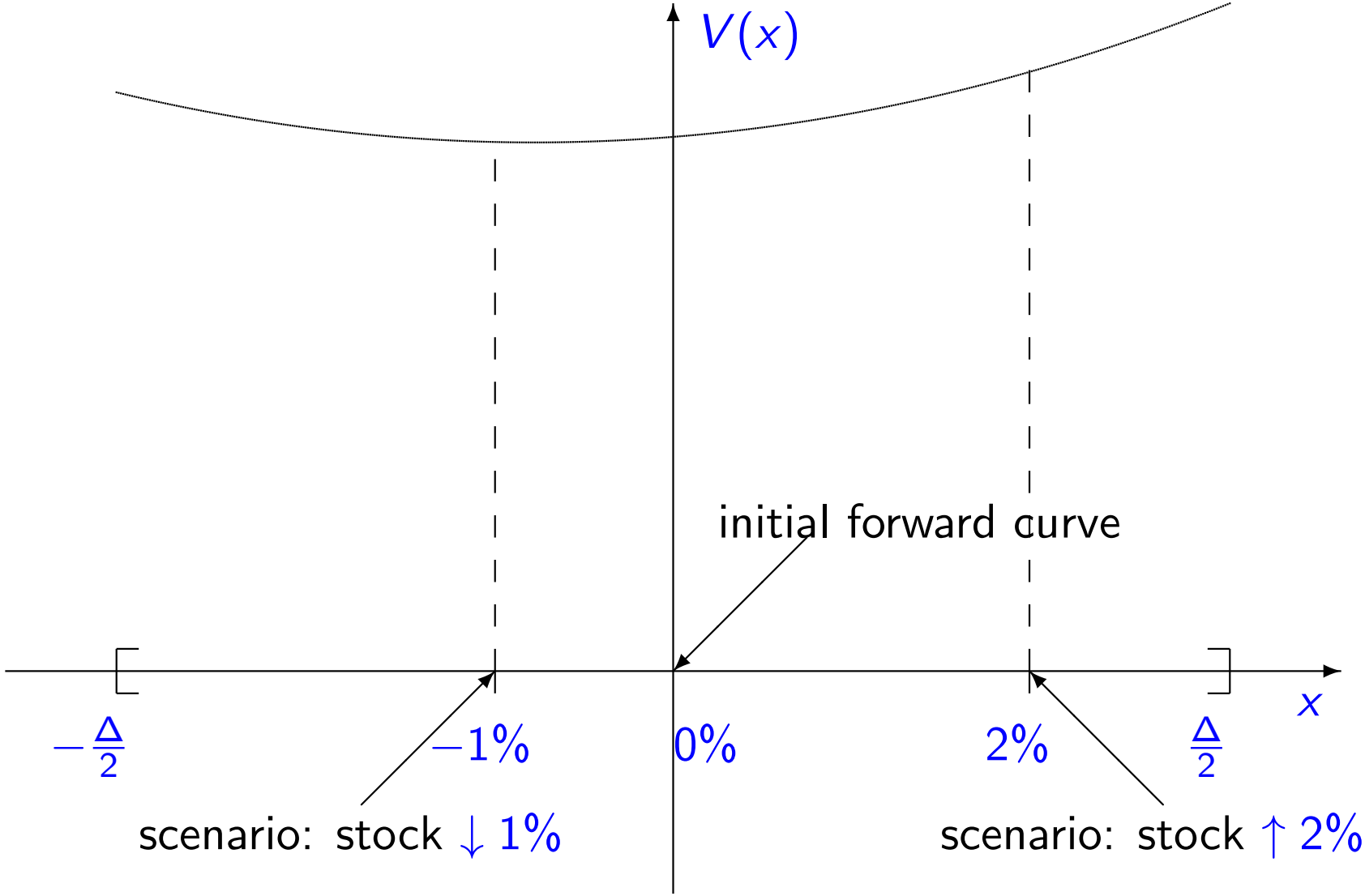
where

$V(x)$: the price of the option corresponding to the scenario that the spot price changes by x percents

x: relative change in the spot price

Δ: width of the interval for relative changes.

Output for Black model



Single asset model in `cfl`

A single asset model in `cfl` is represented by the "universal" class `cfl::AssetModel`.

- The main idea is to "separate" the evaluation of derivatives from the implementations of financial models.
- Creation of basic payoffs (basic `Slice` objects):
  - cash payments,
  - spot prices,
  - forward prices,
  - discount factors.
- Addition of extra state processes to price path dependent derivatives.
- It is constructed from the implementation on a free store of the interface class `cfl::IAssetModel`.
- The interface class `cfl::IAssetModel` and the concrete class `cfl::AssetModel` are related by **pimpl** idiom.

Examples of derivatives

The following derivatives on a single financial asset are evaluated in the project `Examples`:

1. European put
   (`Examples/Src/AssetStdPut.cpp`)
2. American put
   (`Examples/Src/AssetStdAmericanPut.cpp`)
3. Barrier up-or-down-and-out
   (`Examples/Src/AssetStdBarrierUpDownOut.cpp`)
4. Down-and-out American call
   (`Examples/Src/AssetStdDownAndOutAmericanCall.cpp`)
5. Swing option
   (`Examples/Src/AssetStdSwing.cpp`)

Interest rate models

Hull and White model for interest rates
Hull and White model in `cfl`
Interest rate model in `cfl`
Examples of derivatives

Traditional form of Hull and White model

Stationary form:

$$dr_t = (\theta_t - \lambda r_t)dt + \kappa dW_t$$

r_t: short-term rate

λ: mean-reversion rate

κ: volatility of short-term rate

$(\theta_t)_{t\geq 0}$: a function (if $\theta =$ const we get **Vasicek** model)

General form:

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

Calibration:

$(\theta_t)_{t\geq 0}$: discount curve

$(\lambda_t)_{t\geq 0}\&(\kappa_t)_{t\geq 0}$: implied volatility curve

Some notations

$B(s,t)$: discount factor computed at s for maturity t

$r(s,t)$: yield computed at s for maturity t (continuously compounded)

$F(s,t,u)$: forward price of the zero-coupon bond, where

s: current time

t: delivery time for forward

u: maturity of the underlying zero-coupon bond

$$B(s,t) = \exp\left(-(t-s)r(s,t)\right)$$

$$r(s,t) = -\frac{1}{t-s}\ln B(s,t)$$

$$F(s,t,u) = B(s,u)/B(s,t)$$

Hull and White model in terms of forward prices

It is very convenient to write Hull and White model in terms of the evolution of forward prices:

$$dF(s,t,u) = F(s,t,u)(A(u) - A(t))\sigma_s dW_s^t,$$

where

$(A(t))_{t\geq 0}$: initial "shape curve" for changes of yields and discount factors. Convention:

$$A(0) = 0 \quad A'(0) = 1$$

$(\sigma_t)_{t\geq 0}$: instantaneous volatility of forward prices. Relation to volatility of short-term rate: $\kappa_t = \sigma(t)A'(t)$.

W^t: Brownian motion under *forward measure* $\mathbb{P}^t$.

Movements of yields and discount factors

In Hull and White model the movements of yield and discount curves are related as follows:

1. the spot short-term interest rate $r(0) \uparrow 1$ b.p. (basic point)
2. yield $r(0,t) \uparrow \frac{A(t)}{t}$ b.p.
3. discount factor (relative change):

$$\frac{\delta B(0,t)}{B(0,t)} \downarrow A(t) \text{ b.p.}$$

Dynamic of discount curve

In practice,

1. discount factors with longer maturities move "faster" $\Leftrightarrow$ A is increasing $\Leftrightarrow$ $A' > 0$
2. yields with longer maturities move "slower" $\Leftrightarrow$ $\frac{A(t)}{t}$ is decreasing $\Leftrightarrow$ $A' \leq \frac{A(t)}{t}$. This is $\approx$ to the fact that A' is decreasing $\Leftrightarrow$ $A'' < 0$.

If $A' > 0$ and $A'' < 0$, $A'(0) = 1$, then there is a positive function (λ_t) such that

$$A'(t) = \exp(-\int_0^t \lambda_u du).$$

It turns out that (λ_t) is the **mean-reversion rate** that appears in the "traditional" form of Hull and White model.

Input parameters of Hull and White

$(B(0,t))_{t\geq 0}$: initial discount curve

$(A(t))_{t\geq 0}$: initial "shape curve".

$(\Sigma(t)))_{t\geq 0}$: normalized volatility curve:

$$\Sigma(t) = \sqrt{\frac{1}{t}\int_0^t \sigma_u^2 du},$$

Implied volatility at initial time 0 of standard put and call options with delivery s written on zero-coupon bond with maturity t, $s < t$, is given by

$$\Psi(0,s,t) = (A(t) - A(s))\Sigma(s)$$

Stationary implied volatility

The implied volatility function is stationary, i.e.

$$\Psi(s,t,u) = \Psi(0,t-s,u-s)$$

if and only if there are constants κ and λ such that

$$\sigma(t) = \kappa \exp(\lambda t)$$

$$A(t) = \frac{1-\exp(-\lambda t)}{\lambda}$$

$$\Sigma(t) = \kappa\sqrt{\frac{\exp(2\lambda t)-1}{2\lambda t}}$$

$$\Psi(0,t,u) = \kappa\frac{1-\exp(-\lambda(u-t))}{\lambda}\sqrt{\frac{1-\exp(-2\lambda t)}{2\lambda t}}$$

The constants κ and λ appear in the "classical" form for Hull and White model.

Hull and White model in `cfl`

Hull and White model is implemented in the namespace `cfl::HullWhite`:

1. The class `cfl::HullWhite::Data` defines the parameters of the Hull and White model. Recall, that the set of parameters consists of
   - initial time (as year fraction),
   - discount curve,
   - volatility curve,
   - "shape" curve.
2. The functions `cfl::HullWhite::model` implement the *"standard"* interest rate model in the library, namely, the class `cfl::InterestRateModel`.

Output for Hull and White model

It is important (for example, for risk-management) to compute the value of an option in Hull and White model as the function of (minus) **change** in the short-term interest rate:

$$V = (V(x))_{\frac{\Delta}{2} \leq x \leq \frac{\Delta}{2}}$$

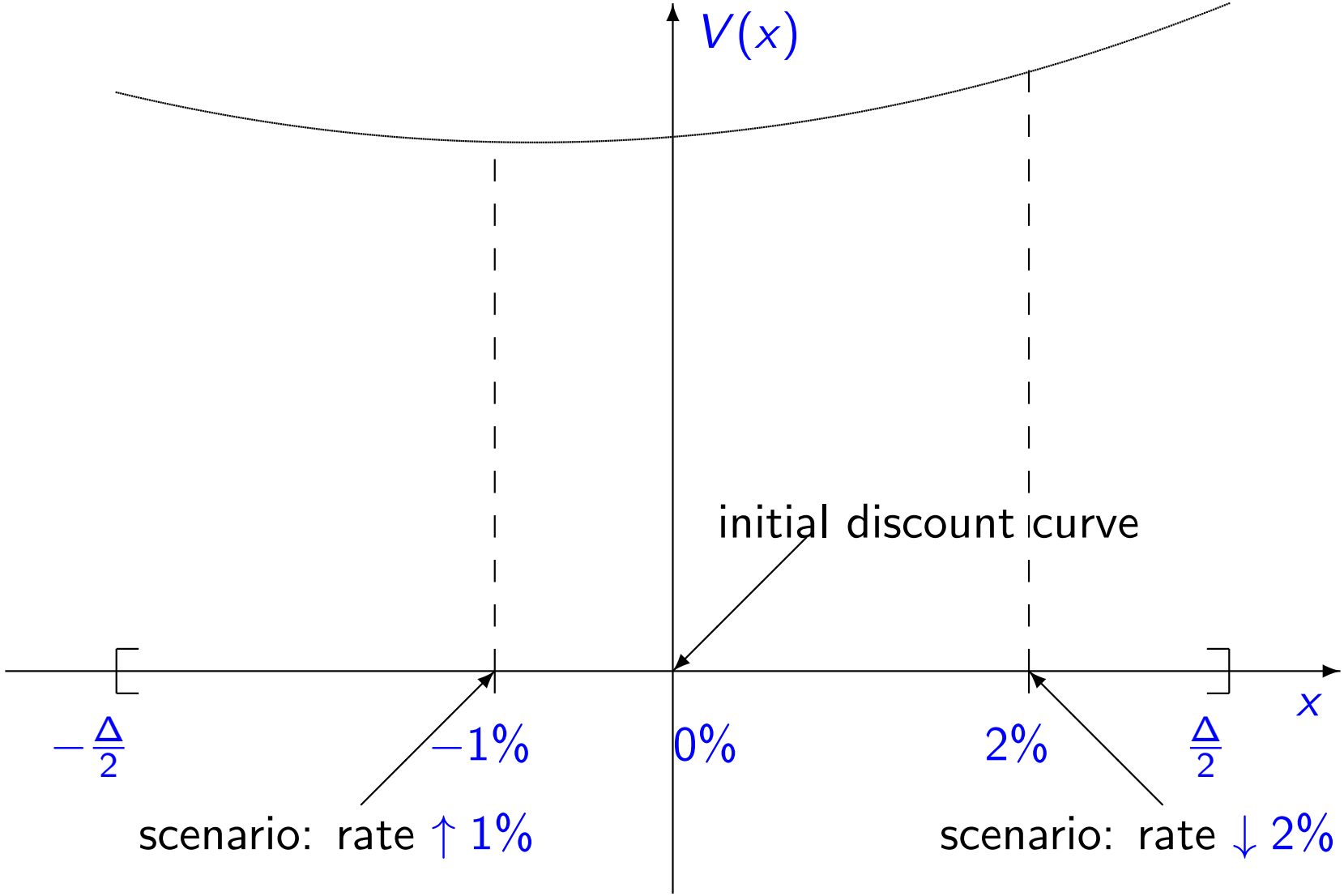
where

$V(x)$: the price of the option corresponding to the scenario that the short-term rate changes by $-x$ percents

x: with sign $-$ the change in the short-term rate

Δ: width of the interval for changes in the short-term rate.

Output for Hull and White model



Interest rate model in `cfl`

An interest rate model in `cfl` is represented by the "universal" class `cfl::InterestRateModel`.

- The main idea is to "separate" the evaluation of derivatives from the implementations of financial models.
- Creation of basic payoffs (basic `cfl::Slice` objects):
  - cash payments,
  - discount factors.
- Addition of extra state processes to price path dependent derivatives.
- It is constructed from the implementation on a free store of the interface class `cfl::IInterestRateModel`.
- The interface class `cfl::IInterestRateModel` and the concrete class `cfl::InterestRateModel` are related by **pimpl** idiom.

Examples of derivatives on interest rates

The following derivatives on a financial asset are evaluated in the project `Examples`:

1. Interest rate cap
   (`Examples/Src/InterestRateStdCap.cpp`)
2. Interest rate swaption
   (`Examples/Src/InterestRateStdSwaption.cpp`)
3. Cancellable collar
   (`Examples/Src/`
   `InterestRateStdCancellableCollar.cpp`)
4. Down and out cap
   (`Examples/Src/InterestRateStdDownOutCap.cpp`)
5. Future on LIBOR
   (`Examples/Src/InterestRateStdFutureOnLibor.cpp`)

Pricing of path-dependent derivatives

Forward start call

Theory

Implementation in `cfl`

Examples of path-dependent derivatives

Motivating example

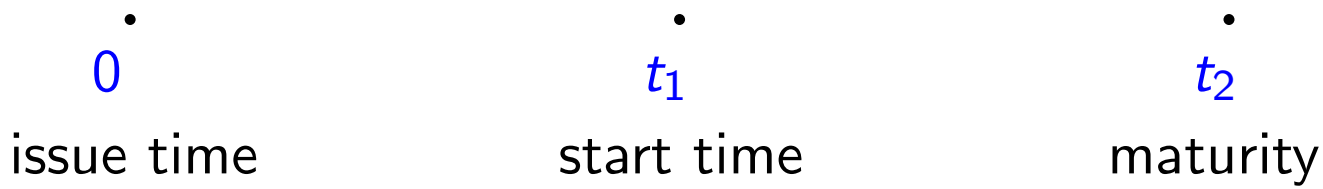
Assume that we have a "standard" implementation of Black model, where the spot price S is a *state process*. In this case, at any time t we can work with random variables in the form:

$$V_t = f(S_t)$$

where $f = f(x)$ is a deterministic function.
Using this implementation of Black model we can price different standard and barrier, European and American options.

Motivating example

Suppose that we have to price *forward start call* option:



which payoff at maturity t_2 is given by

$$V_{t_2} = \max(S_{t_2} - S_{t_1}, 0).$$

Here the *strike* S_{t_1} is determined at t_1.
The payoff function of the forward start call is not "supported" by our standard implementation of Black model.

Motivating example

Solution: extend the dimension of the model

$$S \to (S, Y)$$

where the new component Y is chosen so that

1. (S, Y) is a state process
2. $Y_{t_2} = S_{t_1}$

Then the payoff of the forward start call can be "correctly" expressed in terms of the new 2-dimensional state process (S, Y):

$$V_{t_2} = \max(S_{t_2} - S_{t_1}, 0) = \max(S_{t_2} - Y_{t_2}, 0) = f(S_{t_2}, Y_{t_2})$$

where $f(x, y) = \max(x - y, 0)$.

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

$$\mathcal{X}_t = \{f(X_t) : \ f \text{ is deterministic function } \}, \ t > 0$$

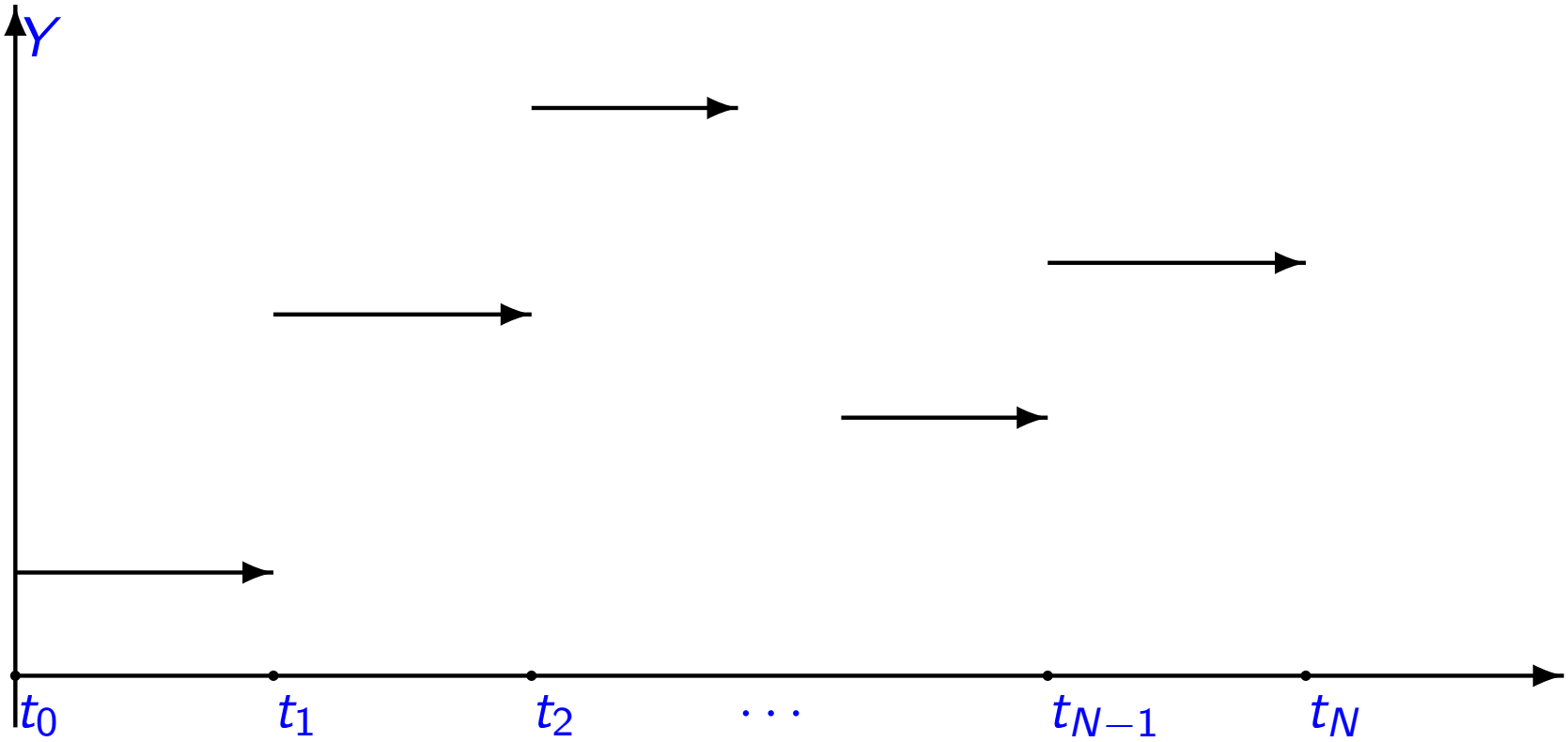
the following operations are implemented:

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** $t_1, \ldots, t_N$:



General framework

Question

When is (X, Y) a state process?

In other words, we want to find the conditions on Y so that for any two times $s < t$ and any payoff at t in the form:

$$V_t = f(X_t, Y_t),$$

where $f = f(x, y)$, the arbitrage-free price of this payoff at s:

$$V_s = \mathcal{R}_s(V_t)$$

has a similar representation:

$$V_s = g(X_s, Y_s)$$

for some $g = g(x, y)$.

Main theorem

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.*

(The value of Y at a reset time is determined by the value of the **original** state process X **at** this time and the value of Y **before** this time).

Example: historical value

In the following examples we assume that the spot price S is a state process.

Example (Historical value)

The process

$$Y_t = 0 \quad t < t_1$$
$$Y_t = S_{t_1} \quad t \geq t_1$$

is useful for the evaluation of **forward start** options.

Example: historical maximum

Example (Historical maximum)

The process

$$Y_t = \max_{t_i \leq t} S_{t_i}$$

is useful for the evaluation of **Lookback** options. We have

$$Y_t = 0 \quad t < t_1$$
$$Y_{t_{i+1}} = \max(Y_{t_i}, S_{t_{i+1}})$$

Example: historical average

Example (Historical average)

The process

$$Y_t = \frac{1}{n(t)} \sum_{i=1}^{n(t)} S_{t_i}$$

where

$$n(t) = \max\{i : \quad t_i \leq t\}$$

is useful for the evaluation of **Asian** options. We have

$$Y_t = 0 \quad t < t_1$$

$$Y_{t_{i+1}} = \frac{1}{i+1}(iY_{t_i} + S_{t_{i+1}})$$

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:** $(G_i)_{1 \le i \le N}$

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

   - 2.3 **initial value:** Y_{t_0} (the value of Y before the first reset time).

Classes in `cfl` library

1. Classes for path dependent processes:
   1.1 Interface class `cfl::IResetValues` (describes reset functions).
   1.2 Concrete class `cfl::PathDependent` (is related to `cfl::IResetValues` through **pimpl** idiom).
2. Classes for model extensions by path dependent processes:
   2.1 Interface class `cfl::IExtend`.
   2.2 Concrete class `cfl::Extended` (is related to `cfl::IExtend` through **pimpl** idiom).

Examples

The following path dependent derivatives are evaluated in project `Examples`:

1. Barrier up-or-down-and-out in single asset model (`Examples/Src/AssetPathBarrierUpDownOut.cpp`)
2. Asian call in single asset model (`Examples/Src/AssetPathAsianCall.cpp`)
3. Savings account in interest rate model (`Examples/Src/InterestRatePathSavingsAccount.cpp`)
4. Put on savings account in interest rate model (`Examples/Src/InterestRatePathPutOnSavingsAccount.cpp`)

Implementation of financial models

Models with identical state processes
Finite differences
Indicators
Approximation
Path-dependent state processes

Models with identical state process

Consider two financial models A and B that have the same

1. maturity T and the forward martingale measure $\mathbb{P}^T$;
2. state process $X = (X_t)_{0 \le t \le T}$.

We call such models A and B **"similar"**.
Assume that

1. the model A has been implemented for the state process X ("old" model)
2. the model B is "new".

Our goal is to implement the "new" model B using the available implementation of the **similar** "old" model A.

Rollback operators

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

For any $s < t$ and any payoff ξ at t the arbitrage-free prices computed at time s in models A and B are related by the following expression:

$$\mathcal{R}_s^B[\xi] = \frac{d^B(s,T)}{d^A(s,T)} \mathcal{R}_s^A \left[\frac{d^A(t,T)}{d^B(t,T)} \xi \right]$$

Rollback operators

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

$$d^A(s, T) = f_s(X_s), \quad d^B(s, T) = g_s(X_s)$$

Denoting $h(x) = f(x)/g(x)$ we deduce that for the payoff ξ at t given by

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

its arbitrage-free price in model B is given by

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

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

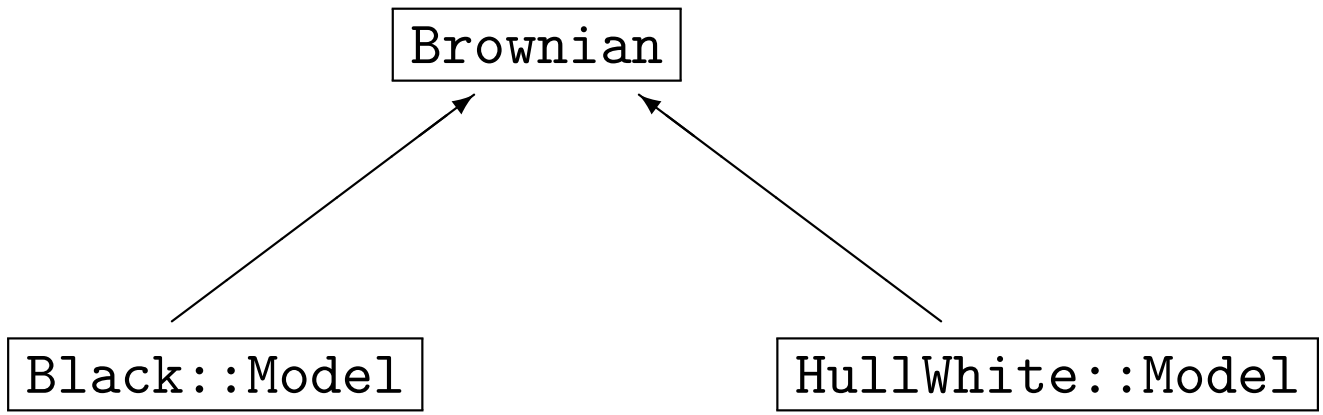
1. Black model
2. Hull and White model
3. Black-Karachinski model
4. Black-Derman-Toy model etc..

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**!



Gaussian conditional expectation

To implement the rollback operator for a financial model with the state process

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

where

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

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

we need to provide numerical implementation of the **operator of conditional expectation** with respect to Gaussian distribution.

Gaussian conditional expectation

Inputs: two times $s < t$ and a deterministic function $f = f(x)$

Output: the deterministic function $g = g(x)$ such that

$$g(X_s) = \mathbb{E}_s[f(X_t)].$$

We have

$$g(x) = \frac{1}{\sqrt{2\pi T}} \int_{-\infty}^{\infty} \exp(-\frac{(x-y)^2}{2T}) f(y) dy,$$

where $T = \int_s^t \sigma^2(u) du$.

Gaussian conditional expectation

For the purpose of numerical implementation we assume that the values of both functions: $f = f(x)$ (input) and $g = g(x)$ (output) are given on a **grid** with $2N + 1$ elements and **step** δx:

$$x_i = i \times \delta x, \quad -N \leq i \leq N.$$

Popular numerical methods:

1. Finite differences
2. Spectral methods (Fast Fourier Transform)

Partial differential equation

The functions $f = f(x)$ (input) and $g = g(x)$ (output) given by

$$g(x) = \frac{1}{\sqrt{2\pi T}} \int_{-\infty}^{\infty} \exp(-\frac{(x-y)^2}{2T}) f(y) dy$$

are also related by the following equation:

Boundary condition:

$$u(x, 0) = f(x)$$

PDE:

$$\frac{\partial u}{\partial t} = \frac{1}{2} \frac{\partial^2 u}{\partial x^2}$$

Result:

$$g(x) = u(x, T).$$

Finite differences

δt: time step

$$\text{(known) } t \longrightarrow t + \delta t \text{ (unknown)}$$

δx: space step

Idea: approximate $\frac{\partial u}{\partial t}$ and $\frac{\partial^2 u}{\partial x^2}$ on the grid of x and t.

Error analysis: shows how "fast" the numerical result converges to the true result

Stability analysis: delivers conditions on δx and δt that guarantee that the scheme is **stable**, that is,

$$\text{Bounded input } \Rightarrow \text{ Bounded output}$$

Approximation for $\frac{\partial u}{\partial t}$

Hereafter we consider the following popular finite difference schemes:

1. Explicit
2. Implicit
3. Crank-Nicolson

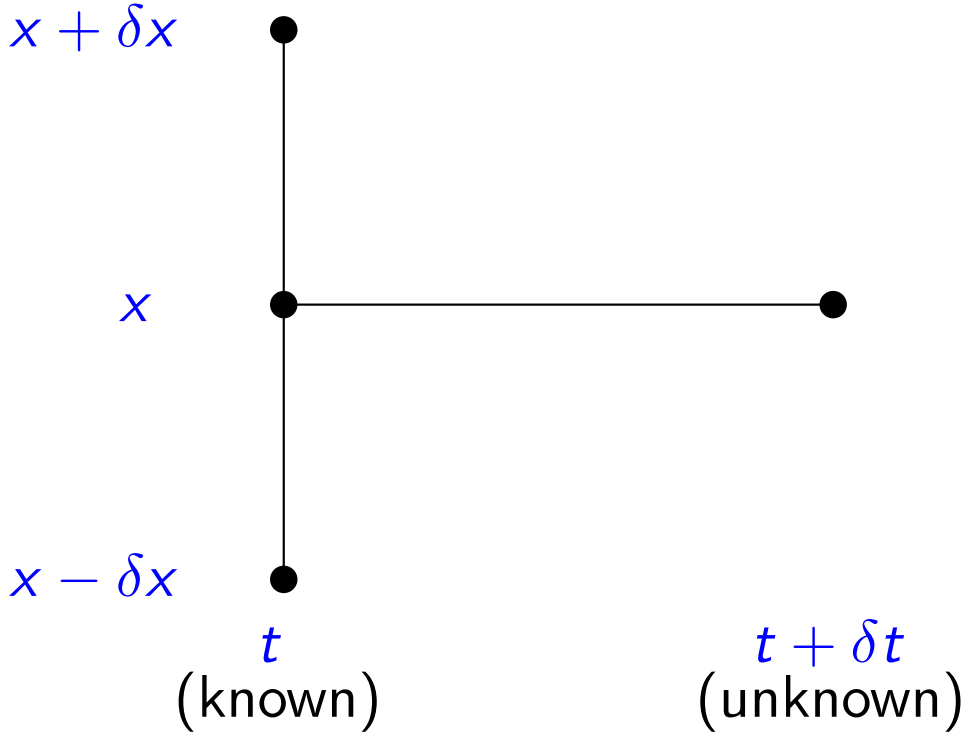
All these schemes share approximation for $\frac{\partial u}{\partial t}$:

$$\frac{\partial u}{\partial t} \approx \frac{u(x, t + \delta t) - u(x, t)}{\delta t}$$

Explicit scheme

Approximation for $\frac{\partial^2 u}{\partial x^2}$ in **explicit** scheme:

$$\frac{\partial^2 u}{\partial x^2} \approx \frac{u(x+\delta x, t) - 2u(x,t) + u(x-\delta x, t)}{(\delta x)^2}$$



Error of explicit scheme

We have

$$u(x, t + \delta t) = (1 - 2p)u(x, t) + p\{u(x - \delta x, t) + u(x + \delta x, t)\}$$

where

$$p = \frac{\delta t}{2(\delta x)^2}.$$

We now perform **error** and **stability** analysis. Using Taylor's expansion for partial derivatives we deduce:

$$\text{Error for explicit scheme} \asymp \delta t + (\delta x)^2$$

Stability of explicit scheme

Definition:

$$\text{Bounded input} \Rightarrow \text{Bounded output}$$

Method: as an input we take

$$f(x) = e^{ikx},$$

where k is an integer. Then the output is

$$u(x,t) = e^{ikx} q_k^{(t/\delta t)}$$

where

$$q_k = (1-2p) + p\{e^{-ik\delta x} + e^{ik\delta x}\} = 1 - 4p \sin^2 \frac{k\delta x}{2}$$

Stability of explicit scheme

We have

$$\text{Bounded output}$$
$$\Updownarrow$$
$$|q_k| \leq 1, \forall k$$
$$\Updownarrow$$
$$p = \frac{\delta t}{2(\delta x)^2} \leq \frac{1}{2}$$
$$\Updownarrow$$
$$\delta t \leq (\delta x)^2$$

The last equality is the **stability condition** for the explicit scheme.

Remarks on explicit scheme

Advantages:
1. Great for not very smooth functions $f(x)$ as inputs
2. This is a **positive** scheme:

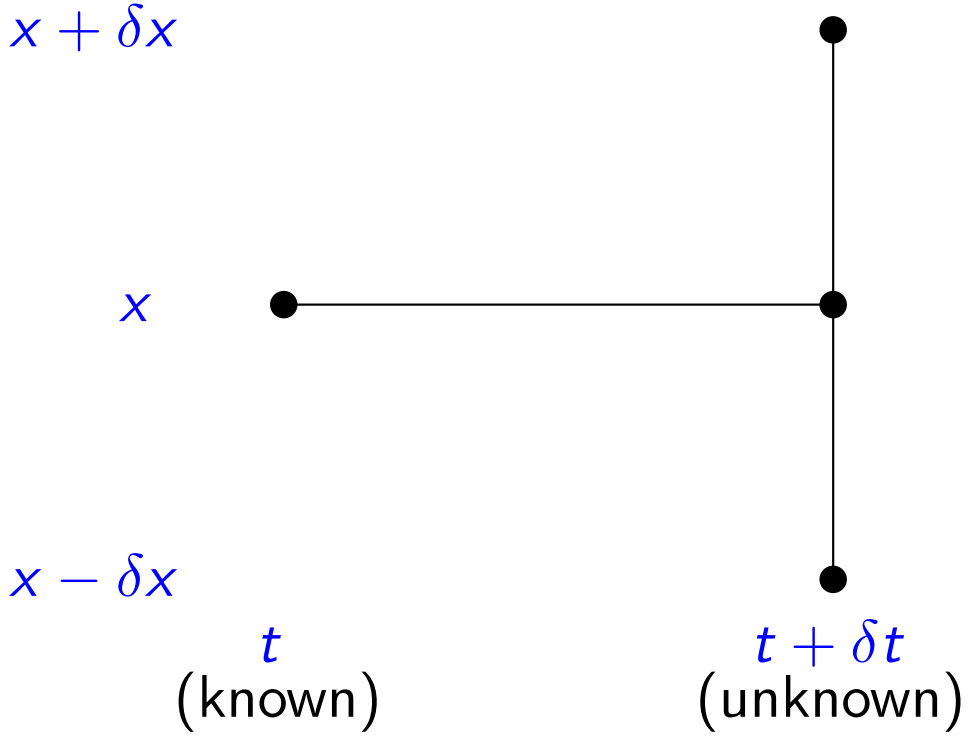
$$\text{Positive input} \Rightarrow \text{Positive output}$$

Disadvantages: might be quite **slow**. Indeed, to double the number of state layers (x's) we have to quadruple the number of time layers.

Implicit scheme

Approximation for $\frac{\partial^2 u}{\partial x^2}$ in **implicit** scheme:

$$\frac{\partial^2 u}{\partial x^2} \approx \frac{u(x+\delta x, t+\delta t) - 2u(x, t+\delta t) + u(x-\delta x, t+\delta t)}{(\delta x)^2}$$



Error of implicit scheme

We have

$$u(x,t) = (1+2p)u(x,t+\delta t) - p\{u(x-\delta x, t+\delta t) + u(x+\delta x, t+\delta t)\}$$

where

$$p = \frac{\delta t}{2(\delta x)^2}.$$

To move one step forward we need to solve a **tridiagonal** system of equations.
Using Taylor's expansion for partial derivatives we deduce:

$$\text{Error for implicit scheme} \asymp \delta t + (\delta x)^2$$

Stability of implicit scheme

As an input we take

$$f(x) = e^{ikx},$$

where k is an integer. Then the output is

$$u(x,t) = e^{ikx} q_k^{(t/\delta t)}$$

where

$$q_k = \frac{1}{1 + 4p \sin^2 \frac{k\delta x}{2}}$$

As $q_k \leq 1$ for any k, the scheme is **stable** (for any p).

Remarks on implicit scheme

Advantages: 1. Great stability

2. No restriction on time step $\Rightarrow$ can be quite fast.

Disadvantages: 1. Not a **positive** scheme:

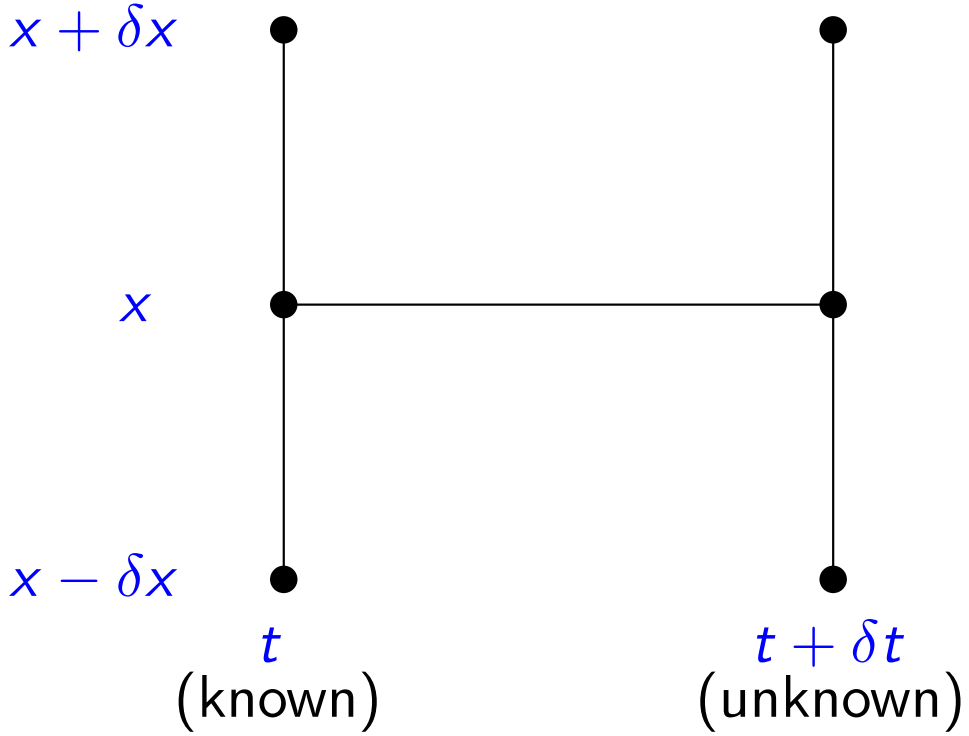
$$\text{Positive input} \not\Rightarrow \text{Positive output}$$

2. Still poor speed of convergence with respect to δt.

Crank-Nicolson scheme

Approximation for $\frac{\partial^2 u}{\partial x^2}$ in **Crank-Nicolson** scheme:

$$\frac{\partial^2 u}{\partial x^2} \approx \frac{1}{2}(\text{“Explicit”} + \text{“Implicit”})$$



Crank-Nicolson scheme

We have

$$\begin{aligned}&(1-2p)u(x,t)+p\{u(x-\delta x,t)+u(x+\delta x,t)\}\\&=(1+2p)u(x,t+\delta t)\\&\quad-p\{u(x-\delta x,t+\delta t)+u(x+\delta x,t+\delta t)\}\end{aligned}$$

where

$$p=\frac{\delta t}{2(\delta x)^2}.$$

As in implicit scheme to move one step forward we need to solve a **tridiagonal** system of equations.

Error of Crank-Nicolson scheme

Using Taylor's expansion for partial derivatives we deduce:

$$\text{Error for Crank-Nicolson} \asymp (\delta t)^2 + (\delta x)^2$$

(Note the term $(\delta t)^2$!).

Stability of Crank-Nicolson scheme

As an input we take

$$f(x) = e^{ikx},$$

where k is an integer. Then the output is

$$u(x,t) = e^{ikx} q_k^{(t/\delta t)}$$

where

$$q_k = \frac{1 - 4p \sin^2 \frac{k\delta x}{2}}{1 + 4p \sin^2 \frac{k\delta x}{2}}$$

As $q_k \leq 1$ for any k, the scheme is **stable** (for any p).

Remarks on Crank-Nicolson scheme

Advantages: 1. Good stability. No restriction on time step.
2. Good speed of convergence with respect to δt.

Disadvantages: Not a **positive** scheme:

$$\text{Positive input} \not\Rightarrow \text{Positive output}$$

Comparison table

Features	Explicit	Implicit	Crank-Nicolson
Positivity	$\star$		
Error			$\star$
Stability		$\star$	

My favorite scheme consists of 3 layers:

1. start with explicit scheme with equal weights (to **smooth** discontinuities).
2. continue with Crank-Nicolson (**fast**)
3. finish with implicit scheme (**stable**).

Implementation in `cfl` library

In `cfl` library the operator of conditional expectation with respect to Gaussian distribution is implemented through the classes

1. `cfl::IGaussRollback`: interface class,
2. `cfl::GaussRollback`: concrete class,

which cooperate by *pimpl* idiom.
Concrete instances are collected in the namespace cfl::NGaussRollback. Currently, the following finite difference methods are implemented: explicit, implicit and Crank-Nicolson.

Integration of discontinuous functions

Basic principle:

$$\text{speed} \asymp \text{smoothness}$$

Consider the **trapezoidal** method of numerical integration on $[-1, 1]$ with the number of steps $2N + 1$:

$$\begin{aligned}\int_{-1}^{1} f(x)dx &= \sum_{-N \le i \le N} \int_{ih}^{(i+1)h} f(x)dx \\ &\approx \sum_{-N \le i \le N} \frac{h}{2}(f(ih) + f((i+1)h))\end{aligned}$$

where $h = 1/N$ is the step of integration.

Case 1: $f(x)$ is smooth

If $f(x)$ is **smooth**, that is, $f'(x)$ is continuous, then

$$\begin{aligned}
&\text{"Error on } [ih,(i+1)h]\text{"} \\
&= \int_{ih}^{(i+1)h} (f(x) - \frac{1}{2}(f(ih) + f((i+1)h))dx \\
&\asymp \int_{ih}^{(i+1)h} (x - ih)^2 dx \asymp h^3
\end{aligned}$$

As the number of intervals $\asymp 1/h$ we deduce that

$$\text{"Total error on } [-1,1]\text{"} \asymp h^2 \asymp 1/N^2$$

Case 2: $f'(x)$ is discontinuous

Assume now that $f(x)$ is **continuous** and $f'(x)$ has a finite number of discontinuities. For example,

$$f(x) = \max(g(x), h(x)),$$

where $g(x)$ and $h(x)$ are smooth functions.

1. If $f'(x)$ is continuous on $[ih, (i+1)h]$ ("good" interval), then as before

$$\text{"Error on } \textit{good}\ [ih, (i+1)h]\text{"} \asymp h^3$$

As the number of "good" intervals $\asymp 1/h$ we deduce that

$$\text{"Total error on } \textit{good}\text{ intervals of } [-1, 1]\text{"} \asymp h^2$$

Case 2: $f'(x)$ is discontinuous

2. If $f'(x)$ is discontinuous on $[ih, (i+1)h]$ ("bad" interval), then

$$\text{"Error on } bad\ [ih, (i+1)h]\text{"}$$
$$= \int_{ih}^{(i+1)h} (f(x) - \frac{1}{2}(f(ih) + f((i+1)h))dx$$
$$\asymp \int_{ih}^{(i+1)h} (x - ih)dx \asymp h^2$$

As the number of "bad" intervals $\asymp 1$ we deduce that

$$\text{"Total error on } bad \text{ intervals of } [-1, 1]\text{"} \asymp h^2$$

Case 2: $f'(x)$ is discontinuous

Hence,

$$
\begin{aligned}
&\text{"Total error on } [-1,1]\text{"} \\
&= \text{"Total error on } \textit{good} \text{ intervals of } [-1,1]\text{"} \\
&\quad + \text{"Total error on } \textit{bad} \text{ intervals of } [-1,1]\text{"} \\
&\asymp h^2 \asymp 1/N^2.
\end{aligned}
$$

Same error as for the smooth case!

Case 3: $f(x)$ is discontinuous

Assume now that $f(x)$ has a finite number of discontinuities. As before we divide all intervals $[ih, (i+1)h]$ into two groups:

1. "Good" intervals, where $f(x)$ is continuous
2. "Bad" intervals, where $f(x)$ is discontinuous.

Our previous analysis implies that

$$\text{"Error on } \textit{good} \text{ intervals of } [-1,1]\text{"} \asymp h^2$$

Case 3: $f(x)$ is discontinuous

We have

$$\begin{aligned}
&\text{"Error on } \textit{bad} \ [ih, (i+1)h]\text{"} \\
&= \int_{ih}^{(i+1)h} (f(x) - \frac{1}{2}(f(ih) + f((i+1)h))dx \\
&\asymp \int_{ih}^{(i+1)h} dx \asymp h
\end{aligned}$$

Hence,

$$\text{"Total error"} = \text{"Error on } \textit{bad} \text{ intervals of } [-1, 1]\text{"} \asymp h$$

(Quite remarkable: one "bad" interval incurs a greater error than all "good" intervals together).

"Smart" indicators

How to increase the speed of convergence of numerical integration for discontinuous functions?

Idea: "smart" representation of indicator functions on the grid:

$$1_{\{x>a\}} \longrightarrow \mathbb{I}^{smart}(ih > a)$$

so that any smooth function $f(x)$

$$\int_{-1}^{1} f(x)I(x > a)dx = \int_{a}^{1} f(x)dx$$

$$\approx \sum_{-N \le i \le N} \frac{h}{2}(f(ih)\mathbb{I}^{smart}(ih > a)$$

$$+ f((i+1)h)\mathbb{I}^{smart}((i+1)h > a)) + O(h^2)$$

We then can use "smart" indicators to model discontinuities.

Example of "smart" indicator

Example

An example of a "smart" indicator if $ih \leq a \leq (i+1)h$:

$$\mathbb{I}^{smart}(kh > a) = 0 \quad k < i$$

$$\mathbb{I}^{smart}(ih > a) = \frac{(i+1)h - a}{2h}$$

$$\mathbb{I}^{smart}((i+1)h > a) = \frac{(i+1)h - a}{2h} + \frac{1}{2}$$

$$\mathbb{I}^{smart}(kh > a) = 1 \quad k > i+1.$$

Numerical example

UP-AND-IN AMERICAN PUT OPTION IN BLACK MODEL

Quality	Smart	Naive	Quality	Smart	Naive
20	**1.1**2388	**1**.09714	720	**1.10673**	**1.10**64
120	**1.10**728	**1.1**0655	820	**1.10673**	**1.10**785
220	**1.106**96	**1.1**0466	920	**1.10672**	**1.10**598
320	**1.106**82	**1.1**0891	1020	**1.10672**	**1.106**32
420	**1.10676**	**1.1**1042	1120	**1.10672**	**1.106**88
520	**1.10674**	**1.10**183	1220	**1.10672**	**1.106**51
620	**1.10674**	**1.10**448	1320	**1.10672**	**1.106**12

Implementation in `cfl` library

In `cfl` library the indicator functions are implemented through

1. the interface class `cfl::IInd`,
2. the concrete class `cfl::Ind`.

These two classes interact with each other by *pimpl* idiom. Concrete implementations are collected in the namespace `cfl::NInd`.

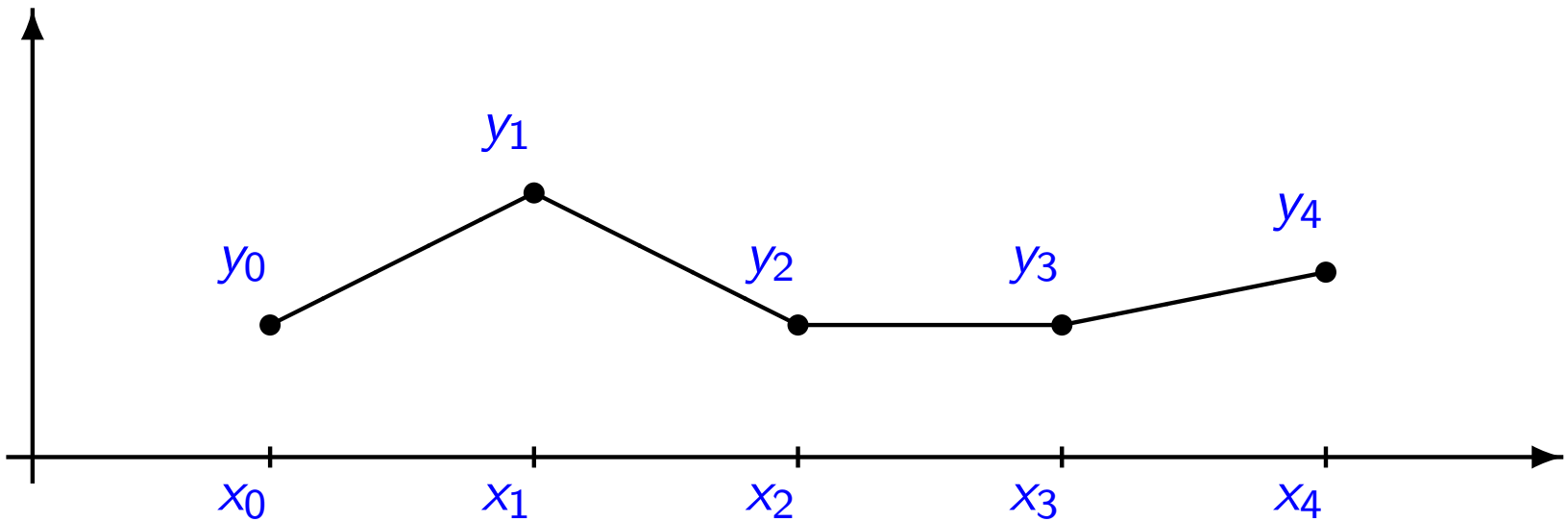
Interpolation and approximation

Basic idea: given

$(x_0, \ldots, x_n)$: arguments

$(y_0, \ldots, y_n)$: values of function $f = f(x)$ $(y_k = f(x_k))$

$\Rightarrow$ **restore** $f = f(x)$ for all x.



Interpolation and approximation

1. In the case of **interpolation** both arguments and values are given as inputs. The methods include
   - 1.1 linear interpolation
   - 1.2 cubic spline interpolation
   - 1.3 polynomial interpolation
2. In the case of **approximation** the arguments are selected first and then the values are computed for these arguments. The methods include
   - 2.1 Chebyshev polynomials
   - 2.2 Trigonometric polynomials

Polynomial interpolation

Assume for simplicity that the arguments belong to $[-1, 1]$:

$$-1 = x_0 < x_1 < \cdots < x_n = 1$$

For any set of values $(y_0, \ldots, y_n)$ there is a unique approximating polynomial $P_n = P_n(x)$ of degree n, that is,

$$P_n(x) = a_0 + a_1 x + \cdots + a_n x^n,$$

and

$$P_n(x_k) = y_k, \quad 0 \leq k \leq n.$$

Polynomial interpolation

Efficient way to compute P_n is based on the representation

$$P_n(x) = \sum_{k=0}^{n} y_k L_k(x),$$

where

$$L_k(x) = \prod_{j=0, j\neq k}^{n} \frac{x - x_j}{x_k - x_j}$$

are *Lagrange polynomials* for which

$$L_k(x_j) = 1_{\{j=k\}}, \quad 0 \leq j \leq k.$$

Polynomial interpolation

Using Taylor's formula one can show that if we use polynomial interpolation for a function $f = f(x)$ then the error is given by

$$|f(x) - P_n(x)| = \frac{f^{(n+1)}(\theta(x))}{(n+1)!} Q(x)$$

where $Q = Q(x)$ is the polynomial of the degree $n + 1$:

$$Q(x) = \prod_{k=0}^{n} (x - x_k)$$

and $\theta(x) \in [-1, 1]$.

"Holy Grail" of Polynomial Approximation

Problem ("Holy Grail" of Polynomial Approximation)

Find the partition

$$-1 = x_0 < x_1 < \cdots < x_n = 1$$

of $[-1, 1]$ *so that*

$$\max_{-1 \le x \le 1} |f(x) - P_n(x)| \to \min .$$

Unfortunately, such *minimax polynomial* P_n for $f = f(x)$ is hard to compute.

Chebyshev approximation

Recall that

$$\max_{-1\le x\le 1} |f(x) - P_n(x)| = \max_{-1\le x\le 1} |\frac{f^{(n+1)}(\theta(x))}{(n+1)!} Q(x)|$$
$$\le \max_{-1\le x\le 1} |\frac{f^{(n+1)}(\theta(x))}{(n+1)!}| \max_{-1\le x\le 1} |Q(x)|$$

where $Q = Q(x)$ is the polynomial of the degree $n+1$:

$$Q(x) = \prod_{k=0}^{n} (x - x_k)$$

Chebyshev approximation

This suggests to look at the following minimization problem:

Problem (Chebyshev approximation)

Find the partition

$$-1 = x_0 < x_1 < \cdots < x_n = 1$$

of $[-1, 1]$ *so that*

$$\max_{-1 \le x \le 1} |Q(x)| \to \min .$$

The solution is given by

$$x_k = \cos\left(\frac{\pi(k + \frac{1}{2})}{n+1}\right), \quad 0 \le k \le n.$$

Chebyshev approximation

The corresponding polynomial approximation is called **Chebyshev approximation**.
The arguments $(x_k)_{0 \le k \le n}$ given above are the roots of the Chebyshev polynomial of the degree $n+1$:

$$T_{n+1}(x) = \cos((n+1)\arccos(x)).$$

$(T_{n+1}(x_k) = 0, \quad 0 \le k \le n)$
We also have the following expression for the error term:

$$Q(x) = \frac{1}{2^n} T_{n+1}(x).$$

Implementation in `cfl`

In `cfl` numerical **approximation** is realized through

1. interface class `cfl::IApprox`
2. concrete class `cfl::Approx`.

Concrete implementations are collected in the namespace `cfl::NApprox`.

Path Dependent state processes

Input: 1. $(t_1, \ldots, t_M)$: event times
2. X: "old" state process
3. $\mathcal{R} = \mathcal{R}^X$: "old" rollback operator (supports X)
4. Y: "new" component for the state process
   4.1 $(s_1, \ldots, s_N)$: reset times (subset of event times)
   4.2 $(G_1, \ldots, G_N)$: reset functions

$$Y_{s_{i+1}} = G_{i+1}(X_{s_{i+1}}, Y_{s_i}).$$

Path Dependent state processes

Output: the rollback operator $\mathcal{R} = \mathcal{R}^{X,Y}$ that supports the state process (X, Y).

In other words given two consecutive *event times* t_i and t_{i+1} and a function $f = f(x, y)$ we need to compute the function $g = g(x, y)$ such that

$$g(X_{t_i}, Y_{t_i}) = \mathcal{R}_{t_i}[f(X_{t_{i+1}}, Y_{t_{i+1}})].$$

Path Dependent state processes

Case 1: t_{i+1} is **not** a *reset time*. Then

$$Y_{t_{i+1}} = Y_{t_i}$$

and

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

"Naive" scheme: for any y we compute

$$g(X_{t_i}, y) = \mathcal{R}_{t_i}[f(X_{t_{i+1}}, y)] = \mathcal{R}^X_{t_i}[f(X_{t_{i+1}}, y)].$$

Path Dependent state processes

"Practical" scheme: choose **approximation method** for the values of Y_{t_i}:

1. Nodes: $(y_1, \ldots, y_K)$
2. Recovery operator: $((y_1, \ldots, y_K), (\theta(y_1), \ldots, \theta(y_K))) \longrightarrow \theta = \theta(y)$.

For every node y_i compute $g_i = g_i(x)$ by

$$g_i(X_{t_i}) = \mathcal{R}_{t_i}[f(X_{t_{i+1}}, y)] = \mathcal{R}^X_{t_i}[f(X_{t_{i+1}}, y)].$$

Then for every x:

$$((y_1, \ldots, y_K), (g_1(x), \ldots, g_K(x)) \overset{\text{recovery}}{\longrightarrow} g(x, y)$$

Path Dependent state processes

Case 2: t_{i+1} is a *reset time*. Then

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

and

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

where

$$h(x, y) = g(x, G_{i+1}(x, y))$$

We follow now the same technique as in the first step.

Path Dependent state processes

In `cfl` library the infrastructure that allows us to add an additional path dependent state process is based on the following classes:

1. `cfl::IExtend`: interface class
2. `cfl::Extended`: concrete class (related to `cfl::IExtend` through **pimpl** idiom).

Implementations of `cfl::Extended` are collected in the namespace `cfl::NExtended`.