QUANTLIB PROJECT

Enable time-dependent steps for binomial tree engines

BOUALILA Karim

BAKKALI Anas

CHEHABI Lina

IDRISSI Aymen

NGUYEN Quang-Nam

TBEZ Hatim

V1.0 **28/02/2018**

TABLE OF CONTENTS

INTRODUCTION

GENERAL PROBLEMATIC

- Currently, the BinomialTree class builds a tree based on constant parameters (risk-free rate, volatility etc.) extracted from the passed stochastic process. In the library, there is also an experimental ExtendedBinomialTree class which uses time-dependent parameters at each step of the tree; however, this currently causes a performance hit because the parameters are recalculated several times.

IN FURTHER DETAIL

- We started by writing a simple main code that priced a European, Bermudan and American option using both the constant parameters BinomialTree class and its extension to non-constant parameters i.e. ExtendedBinominialTree class.

- Since both classes use the same yield/dividends/volatility curves, we expect the option prices to be the same, which they were. We also expected the ExtendedBinomial class to be slower, which it was, in fact it was way slower than we originally expected. The results in the tables below show the runtime differences; up to **11 times for 500 timeSteps**, **37 times for 1500 timeSteps**, **54 times for 3000 timeSteps** which is just huge.

Method	European	Bermudan	American	Time(ms)	Abs diff(ms)
Binomial Jarrow-Rudd	3,845373	4,361986	4,487047	15	79(526%)
Ext Binomial Jarrow-Rudd	3,845373	4,361986	4,487047	94	
Binomial Cox-Ross-Rubinstein	3,843649	4,361042	4,486401	8	39(487%)
Ext Binomial Cox-Ross-Rubinstein	3,843649	4,361042	4,486401	47	
Additive equiprobabilities	3,836045	4,353488	4,478774	16	187(1168%)
Ext Additive equiprobabilities	3,836045	4,353488	4,478774	203	
Binomial Trigeorgis	3,843734	4,36112	4,486475	8	117(1462%)

Ext Binomial Trigeorgis	3,843734	4,36112	4,486475	125	
Binomial Tian	3,845259	4,361725	4,486745	16	77(481%)
Ext Binomial Tian	3,845259	4,361725	4,486745	93	
Binomial Leisen-Reimer	3,841154	4,358383	4,483733	15	124(840%)
Ext Binomial Leisen – Reimer	3,841154	4,358383	4,483733	141	
Binomial Joshi	3,841155	4,358384	4,483734	16	140(875%)
Ext Binomial Joshi	3,841155	4,358384	4,483734	156	

Table 1:Extended Binomial Class performances for steps = 500

Method	European	Bermudan	American	Time(ms)	Abs diff(ms)
Binomial Jarrow-Rudd	3,844638	4,361053	4,486815	62	547(882%)
Ext Binomial Jarrow-Rudd	3,844638	4,361053	4,486815	609	
Binomial Cox-Ross-Rubinstein	3,844324	4,360942	4,486723	31	219(706%)
Ext Binomial Cox-Ross-Rubinstein	3,844324	4,360942	4,486723	250	
Additive equiprobabilities	3,839326	4,356107	4,482098	47	1656(3687%)
Ext Additive equiprobabilities	3,839326	4,356107	4,482098	1703	
Binomial Trigeorgis	3,844351	4,360967	4,486747	31	938(302%)
Ext Binomial Trigeorgis	3,844351	4,360967	4,486747	969	
Binomial Tian	3,844778	4,361308	4,486739	109	610(559%)
`Ext Binomial Tian	3,844778	4,361308	4,486739	719	
Binomial Leisen-Reimer	3,843233	4,360153	4,485691	109	1141(1046%)
Ext Binomial Leisen – Reimer	3,843233	4,360153	4,485691	1250	
Binomial Joshi	3,843233	4,360153	4,485691	109	1048(961%)
Ext Binomial Joshi	3,843233	4,360153	4,485691	1157	

Table 2: Extended Binomial Class performances for steps = 1500

Method	European	Bermudan	American	Time(ms)	Abs diff(ms)
Binomial Jarrow-Rudd	3,844145	4,36088	4,486608	156	2280(1461%)
Ext Binomial Jarrow-Rudd	3,844145	4,36088	4,486608	2436	

Binomial Cox-Ross-Rubinstein	3,844476	4,360885	4,486749	109	828(759%)
Ext Binomial Cox-Ross-Rubinstein	3,844476	4,360885	4,486749	937	
Additive equiprobabilities	3,840351	4,357392	4,483266	125	6766(5412%)
Ext Additive equiprobabilities	3,840351	4,357392	4,483266	6891	
Binomial Trigeorgis	3,844489	4,360897	4,486761	125	3766(3012%)
Ext Binomial Trigeorgis	3,844489	4,360897	4,486761	3891	
Binomial Tian	3,84444	4,36103	4,486658	438	2359(538%)
Ext Binomial Tian	3,84444	4,36103	4,486658	2797	
Binomial Leisen-Reimer	3,843766	4,360494	4,486183	422	4328(1025%)
Ext Binomial Leisen – Reimer	3,843766	4,360494	4,486183	4750	
Binomial Joshi	3,843766	4,360594	4,486183	437	4172(954%)
Ext Binomial Joshi	3,843766	4,360594	4,486183	4609	

Table 3: Extended Binomial Class performances for steps = 3000

- It was clear from the structure of the initial code that the functions with the highest number of calls were going to be driftStep, upStep, dxStep and probUp. These functions take only one argument which is the time t, and yet, the tables below show that they are called up to 1000 times the number of timeSteps. This led us to conclude that these functions recompute the parameters even when they did not change, which is what causes the code to become incredibly slow.

Model	Driftstep	Upstep	dxstep	probup
Ext Binomial Jarrow-Rudd	127527	127527	0	0
Ext Binomial Cox-Ross-Rubinstein	9	0	127542	6
Ext Additive equiprobabilities	510117	127327	0	0
Ext Binomial Trigeorgis	255087	0	255836	6
Ext Binomial Tian	127536	0	0	0
Ext Binomial Leisen –Reimer	255066	0	0	0
Ext Binomial Joshi	255066	0	0	0

Table 4: Function calls for extended binomial class (steps=500)

Model	Driftstep	Upstep	dxstep	probup
Ext Binomial Jarrow-Rudd	1132539	1132539	0	0
Ext Binomial Cox-Ross-Rubinstein	9	0	1132545	6
Ext Additive equiprobabilities	4530165	1132539	0	0
Ext Binomial Trigeorgis	2265111	0	1132548	6
Ext Binomial Tian	1132548	0	0	0
Ext Binomial Leisen –Reimer	2265090	0	0	0
Ext Binomial Joshi	2265090	0	0	0

Table 5: Function calls for extended binomial class (step=1500)

Model	Driftstep	Upstep	dxstep	probup
Ext Binomial Jarrow-Rudd	4514057	4514057	0	0
Ext Binomial Cox-Ross-Rubinstein	9	0	4515063	6
Ext Additive equiprobabilities	18060237	4515057	0	0
Ext Binomial Trigeorgis	9030147	0	4515066	6
Ext Binomial Tian	4515066	0	0	0
Ext Binomial Leisen –Reimer	9030126	0	0	0
Ext Binomial Joshi	9030126	0	0	0

Table 6:Function calls for extended binomial class (steps=3000)

- *Remark:* After running the code, it appears that the probUp function is called at worst 6 times which is negligible, so, no caching machanism will be implemented for this function.

1. SOLUTION

THE CACHING DESIGN PATTERN

- The main idea behind this pattern is to replace the aforementioned functions with functor equivalents that have a memory attribute of type **Cache.** This newly created class stores in a *std::map* container the values of these functions each time they are called, and overloads the () operator so that it checks if the parameter values have changed before computing them, otherwise it would just retrieve the previously computed values from the memory map container. The process of the caching pattern is described below.

(See: http://ljs.academicdirect.org/A08/61_76.pdf for more information.)

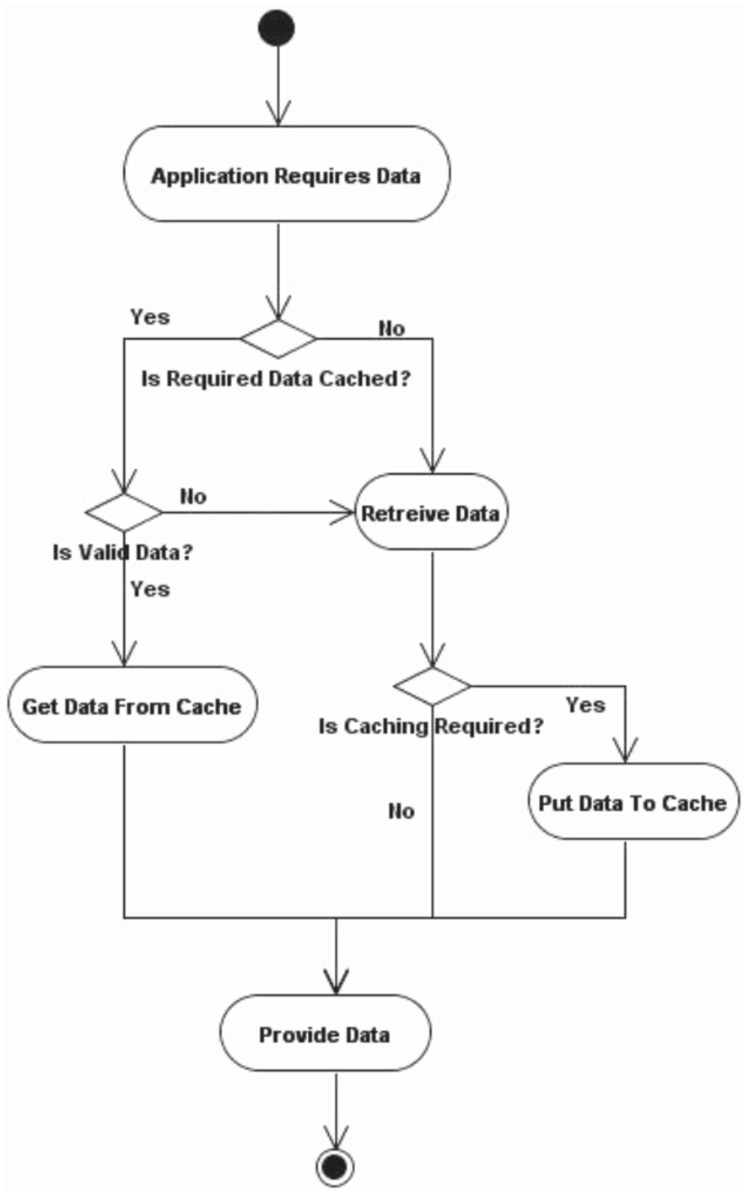


Figure 1: The caching pattern process

IMPLEMENTATION

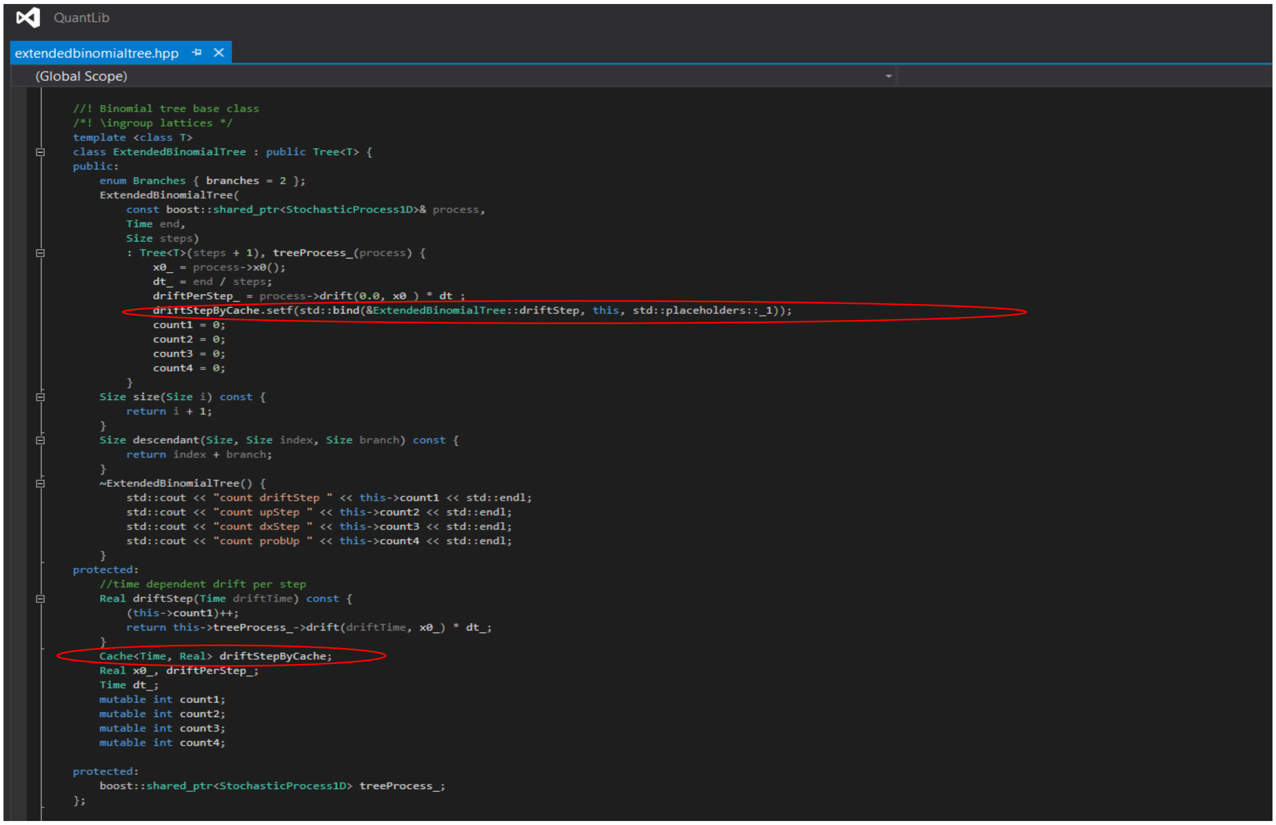
- *To implement this pattern, we needed:*
  - ✓ To create a new template class ***Cache*** which is a functor that takes an *std::function* and creates a caching mechanism for said function. This

implies that the newly written code has to be compiled using the c++11 norm. (To maintain the C++03 compatibility we could use *boost::function*). The code is given below:

```cpp
#ifndef cache_hpp
#define cache_hpp
#include <functional>
#include <map>

template<class T1, class T2>
class Cache
{
public:
    Cache() {}
    Cache(const std::function<T2(T1)> f_){
        f = f_;
    }
    T2 operator() (T1 key) const {
        auto index = memory_map.find(key);
        if (index != memory_map.end())
            return memory_map[key];
        else
            return calculateData(key);
    }

    void setf(const std::function<T2(T1)> f_) {
        f = f_;
    }

    void clear() {
        memory_map.clear;
    }
    void erase(T1 key){
        memory_map.erase(key);
    }

private:
    std::function<T2(T1)> f;
    mutable std::map<T1, T2> memory_map;

    T2 calculateData(T1 key) const {
        put(key, f(key));
        return memory_map[key];
    }

    template<class T3>
    void put(T1 key, T3 value) const {
        memory_map[key] = static_cast<T2>(value);
    }
};
#endif
```

Figure 2: Cache.hpp class

- ✓ To modify the concerned classes by adding the *Cache* objects and modify the constructors to instantiate them. An example is given below for the driftStep caching mechanism, the same modifications were applied to the other classes:




```cpp
//! Binomial tree base class
/*! \ingroup lattices */
template <class T>
class ExtendedBinomialTree : public Tree<T> {
public:
    enum Branches { branches = 2 };
    ExtendedBinomialTree(
        const boost::shared_ptr<StochasticProcess1D>& process,
        Time end,
        Size steps)
        : Tree<T>(steps + 1), treeProcess_(process) {
            x0_ = process->x0();
            dt_ = end / steps;
            driftPerStep_ = process->drift(0.0, x0_) * dt_;
            driftStepByCache.setf(std::bind(&ExtendedBinomialTree::driftStep, this, std::placeholders::_1));
            count1 = 0;
            count2 = 0;
            count3 = 0;
            count4 = 0;
        }
    Size size(Size i) const {
        return i + 1;
    }
    Size descendant(Size, Size index, Size branch) const {
        return index + branch;
    }
    ~ExtendedBinomialTree() {
        std::cout << "count driftStep " << this->count1 << std::endl;
        std::cout << "count upStep " << this->count2 << std::endl;
        std::cout << "count dxStep " << this->count3 << std::endl;
        std::cout << "count probUp " << this->count4 << std::endl;
    }
protected:
    //time dependent drift per step
    Real driftStep(Time driftTime) const {
        (this->count1)++;
        return this->treeProcess_->drift(driftTime, x0_) * dt_;
    }
    Cache<Time, Real> driftStepByCache;
    Real x0_, driftPerStep_;
    Time dt_;
    mutable int count1;
    mutable int count2;
    mutable int count3;
    mutable int count4;

protected:
    boost::shared_ptr<StochasticProcess1D> treeProcess_;
};
```

Figure 3: Caching mechanism for the driftStep function (1)

```cpp
//! Base class for equal probabilities binomial tree
/*! \ingroup lattices */
template <class T>
class ExtendedEqualProbabilitiesBinomialTree
    : public ExtendedBinomialTree<T> {
public:
    ExtendedEqualProbabilitiesBinomialTree(
        const boost::shared_ptr<StochasticProcess1D>& process,
        Time end,
        Size steps)
        : ExtendedBinomialTree<T>(process, end, steps) {}
    virtual ~ExtendedEqualProbabilitiesBinomialTree() {}

    Real underlying(Size i, Size index) const {
        Time stepTime = i*this->dt_;
        BigInteger j = 2 * BigInteger(index) - BigInteger(i);
        // exploiting the forward value tree centering
        return this->x0_*std::exp(i*this->driftStepByCache(stepTime) + j*this->upStepByCache(stepTime));
    }

    Real probability(Size, Size, Size) const { return 0.5; }
protected:
    //the tree dependent up move term at time stepTime
    virtual Real upStep(Time stepTime) const = 0;
    Cache<Time, Real> upStepByCache;
    Real up_;
};
```

Figure 4: Caching mechanism for the driftStep function (2)

2.RESULTS

PERFORMANCE TEST WITH CONSTANT PARAMETERS BSM MODEL

- The performance tests were implemented for the following trees models:
  - ✓ Binomial Jarrow-Rudd
  - ✓ Cox-Ross-Rubinstein
  - ✓ Additive equiprobabilities
  - ✓ Binomial trigeoris
  - ✓ Binomial Tian
  - ✓ Binomial Leisen-Reimer
  - ✓ Binomial Joshi
- We also tested the impact of choosing different time steps on the code's runtime, the time steps we chose are:
  - ✓ 500
  - ✓ 1500
  - ✓ 3000

Let's take a look at the result:

MMethod	European	Bermudan	American	Time(ms)	Abs diff(ms)
Ext Binomial Jarrow-Rudd	3,845373	4,361986	4,487047	94	78(487%)
Cached Ext Binomial Jarrow-Rudd	3,845373	4,361986	4,487047	16	
Ext Binomial Cox-Ross-Rubinstein	3,843649	4,361042	4,486401	47	31(193%)
Cached Ext Binomial Cox-Ross-Rubinstein	3,843649	4,361042	4,486401	16	
Ext Additive equiprobabilities	3,836045	4,353488	4,478774	203	187(1168%)
Cached Ext Additive equiprobabilities	3,836045	4,353488	4,478774	16	
Ext Binomial Trigeorgis	3,843734	4,36112	4,486475	125	109(681%)
Cached Ext Binomial Trigeorgis	3,843734	4,36112	4,486475	16	
Ext Binomial Tian	3,845259	4,361725	4,486745	93	30(187%)
Cached Ext Binomial Tian	3,845259	4,361725	4,486745	63	
Ext Binomial Leisen-Reimer	3,841154	4,358383	4,483733	141	79(127%)
Cached Ext Binomial Leisen –Reimer	3,841154	4,358383	4,483733	62	

Ext Binomial Joshi	3,841155	4,358384	4,483734	156	93(147%)
Cached Ext Binomial Joshi	3,841155	4,358384	4,483734	63	

Table 7: Extended Binomial Class performances for steps = 500

Method	European	Bermudan	American	Time(ms)	Abs diff(ms)
Ext Binomial Jarrow-Rudd	3,844638	4,361053	4,486815	609	468(331%)
Cached Ext Binomial Jarrow-Rudd	3,844638	4,361053	4,486815	141	
Ext Binomial Cox-Ross-Rubinstein	3,844324	4,360942	4,486723	250	156(165%)
Cached Ext Binomial Cox-Ross-Rubinstein	3,844324	4,360942	4,486723	94	
Ext Additive equiprobabilities	3,839326	4,356107	4,482098	708	552(354%)
Cached Ext Additive equiprobabilities	3,839326	4,356107	4,482098	156	
Ext Binomial Trigeorgis	3,844351	4,360967	4,486747	976	851(680%)
Cached Ext Binomial Trigeorgis	3,844351	4,360967	4,486747	125	
Ext Binomial Tian	3,844778	4,361308	4,486739	729	310(73%)
Cached Ext Binomial Tian	3,844778	4,361308	4,486739	419	
Ext Binomial Leisen-Reimer	3,843233	4,360153	4,485691	1250	678(118%)
Cached Ext Binomial Leisen –Reimer	3,843233	4,360153	4,485691	572	
Ext Binomial Joshi	3,843233	4,360153	4,485691	1157	610(111%)
Cached Ext Binomial Joshi	3,843233	4,360153	4,485691	547	

Table 8: Extended Binomial Class performances for steps = 1500

Method	European	Bermudan	American	Time(ms)	Abs diff(ms)
Ext Binomial Jarrow-Rudd	3,844145	4,36088	4,486608	2436	1774(267%)
Cached Ext Binomial Jarrow-Rudd	3,844145	4,36088	4,486608	662	
Ext Binomial Cox-Ross-Rubinstein	3,844476	4,360885	4,486749	950	544(135%)
Cached Ext Binomial Cox-Ross-Rubinstein	3,844476	4,360885	4,486749	406	
Ext Additive equiprobabilities	3,840351	4,357392	4,483266	6891	6214(918%)
Cached Ext Additive equiprobabilities	3,840351	4,357392	4,483266	677	
Ext Binomial Trigeorgis	3,844489	4,360897	4,486761	3891	3470(924%)
Cached Ext Binomial Trigeorgis	3,844489	4,360897	4,486761	421	
Ext Binomial Tian	3,84444	4,36103	4,486658	2819	1186(73%)
Cached Ext Binomial Tian	3,84444	4,36103	4,486658	1633	
Ext Binomial Leisen-Reimer	3,843766	4,360494	4,486183	4750	2438(105%)
Cached Ext Binomial Leisen –Reimer	3,843766	4,360494	4,486183	2313	
Ext Binomial Joshi	3,843766	4,360594	4,486183	4609	2421(110%)
Cached Ext Binomial Joshi	3,843766	4,360594	4,486183	2188	

Table 9: Extended Binomial Class performances for steps = 3000

❖ It is clear from the tables above that the modified class outperforms by far the initial one, we can sum up the performance gain in the following point:

- For 500: it went from a total of 859ms to 25ms which is almost 3,4 times faster.
- For 1500: it went from a total of 6,313 s to 2,054 s which is almost 3 times faster.
- For 3000: it went from a total of 26,346 s to 8,3 s which is almost 3,2 times faster.

Model	Driftstep	Upstep	dxstep	probup
Cached Ext Binomial Jarrow-Rudd	510	510	0	0
Cached Ext Binomial Cox-Ross-Rubinstein	3	0	512	3
Cached Ext Additive equiprobabilities	512	510	0	0
Cached Ext Binomial Trigeorgis	512	0	512	3
Cached Ext Binomial Tian	512	0	0	0
Cached Ext Binomial Leisen –Reimer	512	0	0	0
Cached Ext Binomial Joshi	512	0	0	0

Table 10: Function calls for extended binomial class (steps=500)

Model	Driftstep	Upstep	dxstep	probup
Cached Ext Binomial Jarrow-Rudd	1509	1509	0	0
Cached Ext Binomial Cox-Ross-Rubinstein	3	0	1512	3
Cached Ext Additive equiprobabilities	1512	1510	0	0
Cached Ext Binomial Trigeorgis	1512	0	1512	3
Cached Ext Binomial Tian	1512	0	0	0
Cached Ext Binomial Leisen –Reimer	1512	0	0	0
Cached Ext Binomial Joshi	1512	0	0	0

Table 11: Function calls for extended binomial class (step=1500)

Model	Driftstep	Upstep	dxstep	probup
Cached Ext Binomial Jarrow-Rudd	3010	3010	0	0
Cached Ext Binomial Cox-Ross-Rubinstein	3	0	3012	3
Cached Ext Additive equiprobabilities	3012	3010	0	0
Cached Ext Binomial Trigeorgis	3012	0	3012	3
Cached Ext Binomial Tian	3012	0	0	0
Cached Ext Binomial Leisen –Reimer	3012	0	0	0
Cached Ext Binomial Joshi	3012	0	0	0

- It is clear from the tables above that we reduced the number of calls to the functions driftStep, upStep and dxStep. It is now very close to the number of timeSteps which is to be expected, because we used time as the key to the memory map of the *Cache* class.

PERFORMANCE TEST WITH NON CONSTANT PARAMETERS BSM MODEL

- In this part, we had to build a BSM with non-constant parameters. To do that we replaced the flat yield and volatility term structures with non-constant ones that we initialized arbitrarily. We used the *zeroCurve* and *blackVarianceCurve* classes to achieve that, the implementation is given in the code below.

- To switch between the flat term structures and non-constant ones, one should use the right TS while building the *blackScholesMerton* process.

```cpp
(Global Scope)                                                    main(int, char *[])

        // non constant yield curve
        std::vector<Date> dates(3);
        dates[0] = settlementDate;
        dates[1] = settlementDate + Period(1, Years);
        dates[2] = settlementDate + Period(2, Years);
        std::vector<Rate> rates(3);
        rates[0] = 0.05;
        rates[1] = 0.06;
        rates[2] = 0.075;
        Handle<YieldTermStructure> TermStructure(
            boost::shared_ptr<YieldTermStructure>(
            new ZeroCurve(dates, rates, dayCounter)));

        //flat divs curve
        Handle<YieldTermStructure> flatDividendTS(
            boost::shared_ptr<YieldTermStructure>(
            new FlatForward(settlementDate, dividendYield, dayCounter)));

        //flat vol curve
        Handle<BlackVolTermStructure> flatVolTS(
            boost::shared_ptr<BlackVolTermStructure>(
            new BlackConstantVol(settlementDate, calendar, volatility,
            dayCounter)));

        //non constant vol curve
        std::vector<Date> voldates(3);
        voldates[0] = settlementDate + Period(4, Months);
        voldates[1] = settlementDate + Period(8, Months);
        voldates[2] = settlementDate + Period(1, Years);
        std::vector<Volatility> volatilities(3);
        volatilities[0] = 0.025;
        volatilities[1] = 0.036;
        volatilities[2] = 0.0475;
        Handle<BlackVolTermStructure> VolTS(
            boost::shared_ptr<BlackVolTermStructure>(
            new BlackVarianceCurve(settlementDate,voldates, volatilities,
            dayCounter)));

        //payoff and process
        boost::shared_ptr<StrikedTypePayoff> payoff(
            new PlainVanillaPayoff(type, strike));

        boost::shared_ptr<BlackScholesMertonProcess> bsmProcess(
            new BlackScholesMertonProcess(underlyingH, flatDividendTS,
            flatTermStructure, flatVolTS)); // use TermStructure, VolTS instead for non constant parameters
```

This is where you can switch between flat and non-constant term structures

Figure 5: flat and non-constant parameters term structures code

- The performance tests were implemented for the same models as 0. Let's take a look at the result:

Method	European	Bermudan	American	Time(ms)	Abs diff(ms)
Ext Binomial Jarrow-Rudd	1,829030	3,399786	4	89	58(187%)
Cached Ext Binomial Jarrow-Rudd	1,829030	3,399786	4	31	
Ext Binomial Cox-Ross-Rubinstein	1,828271	3,399813	4	54	39(260%)
Cached Ext Binomial Cox-Ross-Rubinstein	1,828271	3,399813	4	15	
Ext Additive equiprobabilities	1,818596	3,400353	4	217	202(1346%)
Cached Ext Additive equiprobabilities	1,818596	3,400353	4	15	
Ext Binomial Trigeorgis	1,828885	3,399782	4	128	105(456%)
Cached Ext Binomial Trigeorgis	1,828885	3,399782	4	23	
Ext Binomial Tian	1,829167	3,399786	4	95	48(102%)

Cached Ext Binomial Tian	1,829167	3,399786	4	47	
Ext Binomial Leisen-Reimer	1,831999	3,400872	4	156	93(147%)
Cached Ext Binomial Leisen –Reimer	1,831999	3,400872	4	63	
Ext Binomial Joshi	1,131999	3,400772	4	156	86(123%)
Cached Ext Binomial Joshi	1,131999	3,400772	4	70	

Table 12: Extended Binomial Class performances for steps = 500

Method	European	Bermudan	American	Time(ms)	Abs diff(ms)
Ext Binomial Jarrow-Rudd	1,829156	3,399788	4	625	468(298%)
Cached Ext Binomial Jarrow-Rudd	1,829156	3,399788	4	157	
Ext Binomial Cox-Ross-Rubinstein	1,828839	3,399797	4	250	141(129%)
Cached Ext Binomial Cox-Ross-Rubinstein	1,828839	3,399797	4	109	
Ext Additive equiprobabilities	1,823145	3,400111	4	1703	1531(890%)
Cached Ext Additive equiprobabilities	1,823145	3,400111	4	172	
Ext Binomial Trigeorgis	1,829042	3,399787	4	953	858(903%)
Cached Ext Binomial Trigeorgis	1,829042	3,399787	4	95	
Ext Binomial Tian	1,829280	3,399788	4	710	303(74%)
Cached Ext Binomial Tian	1,829280	3,399788	4	407	
Ext Binomial Leisen-Reimer	1,830138	3,400151	4	1183	605(104%)
Cached Ext Binomial Leisen –Reimer	1,830138	3,400151	4	578	

Ext Binomial Joshi	1,830138	3,400151	4	1141	578(102%)
Cached Ext Binomial Joshi	1,830138	3,400151	4	563	

Table 13: Extended Binomial Class performances for steps = 500

Method	European	Bermudan	American	Time(ms)	Abs diff(ms)
Ext Binomial Jarrow-Rudd	1,829225	3,399789	4	2402	1730(257%)
Cached Ext Binomial Jarrow-Rudd	1,829225	3,399789	4	672	
Ext Binomial Cox-Ross-Rubinstein	1,829076	3,399794	4	938	532(231%)
Cached Ext Binomial Cox-Ross-Rubinstein	1,829076	3,399794	4	406	
Ext Additive equiprobabilities	1,824915	3,400016	4	6703	6047(921%)
Cached Ext Additive equiprobabilities	1,824915	3,400016	4	656	
Ext Binomial Trigeorgis	1,828174	3,399788	4	3817	3395(804%)
Cached Ext Binomial Trigeorgis	1,828174	3,399788	4	422	
Ext Binomial Tian	1,829219	3,399789	4	2750	1138(70%)
Cached Ext Binomial Tian	1,829219	3,399789	4	1612	
Ext Binomial Leisen-Reimer	1,82968	3,39997	4	4586	2273(98%)
Cached Ext Binomial Leisen –Reimer	1,82968	3,39997	4	2313	
Ext Binomial Joshi	1,829680	3,39997	4	4496	2340(108%)
Cached Ext Binomial Joshi	1,829680	3,39997	4	2156	

Table 14: Extended Binomial Class performances for steps = 3000

- The prices changed as expected since we used different yield and volatility structures.

- The runtime absolute difference between the cached and the extended binomial classes using non-constant parameters are:

  - ✓ For 500: it went from a total of 895ms to 264ms which is almost 3,4 times faster.
  - ✓ For 1500: it went from a total of 6,565 s to 2,081 s which is almost 3,15 times faster.
  - ✓ For 3000: it went from a total of 25,699 s to 8,237 s which is almost 3,12 times faster.
- Which is very very close to the performance gains we obtained when using the constant parameter BSM model, which was what we expected.