Quantlib Project : Enable time-dependent steps for binomial tree engines

CHHOA André
HAJJAM Tarik
TRIKI Anis
XIANG Zhun

February 2017

Under the supervision of Luigi Ballabio

Contents

1 Introduction **3**
1.1 Subject . . . 3
1.2 Analysis of the problematic . . . 3

2 Solution **6**
2.1 A Design Pattern: the Cache . . . 6
2.2 Implementation . . . 7

3 Result **8**
3.1 Performance tests . . . 8
3.2 Reduction of the Complexity . . . 12

1 Introduction

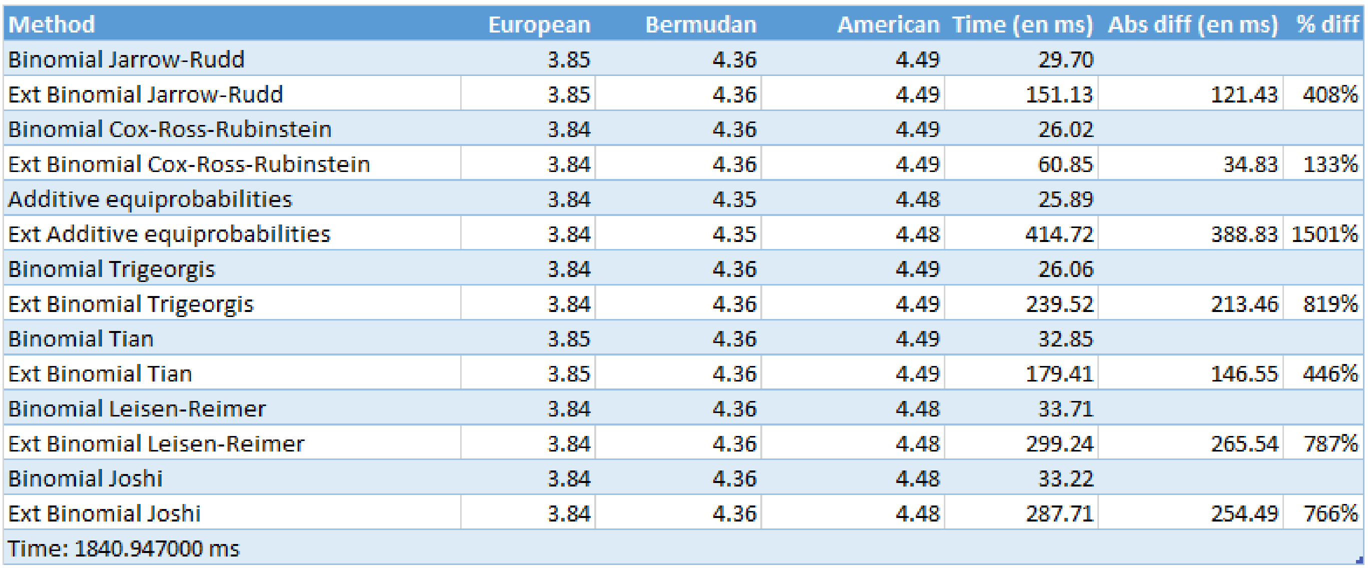
1.1 Subject

In the Quantlib library, there is an ExtendedBinomialTree class which uses time-dependent parameters at each step of the tree. However, the performances are pretty bad because the parameters are recalculated several times.

The aim of this project is to find a way to optimize this class and get better performances.

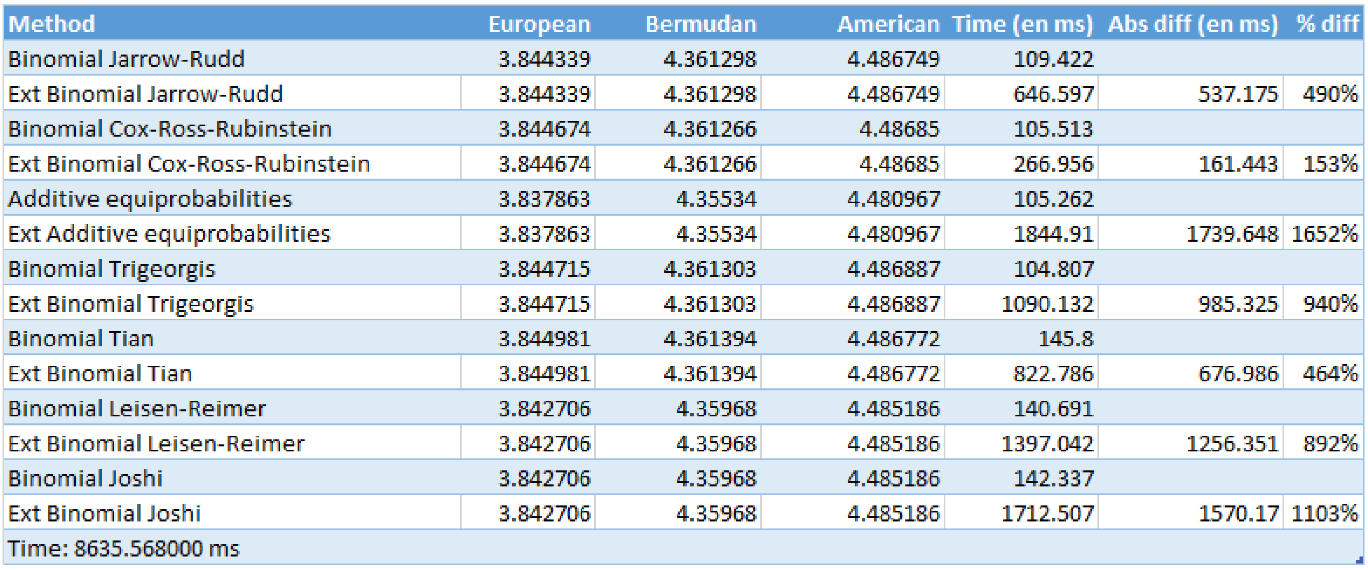
1.2 Analysis of the problematic

The main issue with the current class is that the time performances for the calculations are too bad. It is not a surprise that the extended binomial tree will take longer to execute than the usual binomial tree as it has to calculate the different parameters more than once. However, according to the test we did with different number of time steps the difference is huge :



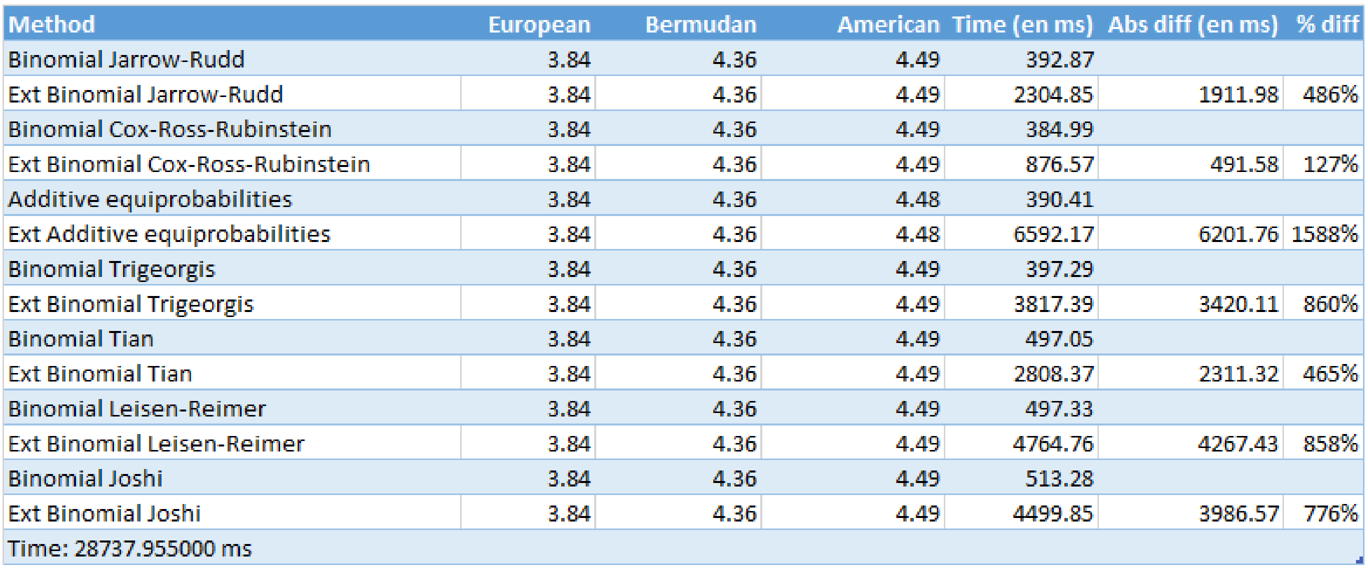
Method	European	Bermudan	American	Time (en ms)	Abs diff (en ms)	% diff
Binomial Jarrow-Rudd	3.85	4.36	4.49	29.70		
Ext Binomial Jarrow-Rudd	3.85	4.36	4.49	151.13	121.43	408%
Binomial Cox-Ross-Rubinstein	3.84	4.36	4.49	26.02		
Ext Binomial Cox-Ross-Rubinstein	3.84	4.36	4.49	60.85	34.83	133%
Additive equiprobabilities	3.84	4.35	4.48	25.89		
Ext Additive equiprobabilities	3.84	4.35	4.48	414.72	388.83	1501%
Binomial Trigeorgis	3.84	4.36	4.49	26.06		
Ext Binomial Trigeorgis	3.84	4.36	4.49	239.52	213.46	819%
Binomial Tian	3.85	4.36	4.49	32.85		
Ext Binomial Tian	3.85	4.36	4.49	179.41	146.55	446%
Binomial Leisen-Reimer	3.84	4.36	4.48	33.71		
Ext Binomial Leisen-Reimer	3.84	4.36	4.48	299.24	265.54	787%
Binomial Joshi	3.84	4.36	4.48	33.22		
Ext Binomial Joshi	3.84	4.36	4.48	287.71	254.49	766%
Time: 1840.947000 ms						

Figure 1: Current Class performances for steps = 500



Method	European	Bermudan	American	Time (en ms)	Abs diff (en ms)	% diff
Binomial Jarrow-Rudd	3.844339	4.361298	4.486749	109.422		
Ext Binomial Jarrow-Rudd	3.844339	4.361298	4.486749	646.597	537.175	490%
Binomial Cox-Ross-Rubinstein	3.844674	4.361266	4.48685	105.513		
Ext Binomial Cox-Ross-Rubinstein	3.844674	4.361266	4.48685	266.956	161.443	153%
Additive equiprobabilities	3.837863	4.35534	4.480967	105.262		
Ext Additive equiprobabilities	3.837863	4.35534	4.480967	1844.91	1739.648	1652%
Binomial Trigeorgis	3.844715	4.361303	4.486887	104.807		
Ext Binomial Trigeorgis	3.844715	4.361303	4.486887	1090.132	985.325	940%
Binomial Tian	3.844981	4.361394	4.486772	145.8		
Ext Binomial Tian	3.844981	4.361394	4.486772	822.786	676.986	464%
Binomial Leisen-Reimer	3.842706	4.35968	4.485186	140.691		
Ext Binomial Leisen-Reimer	3.842706	4.35968	4.485186	1397.042	1256.351	892%
Binomial Joshi	3.842706	4.35968	4.485186	142.337		
Ext Binomial Joshi	3.842706	4.35968	4.485186	1712.507	1570.17	1103%
Time: 8635.568000 ms						

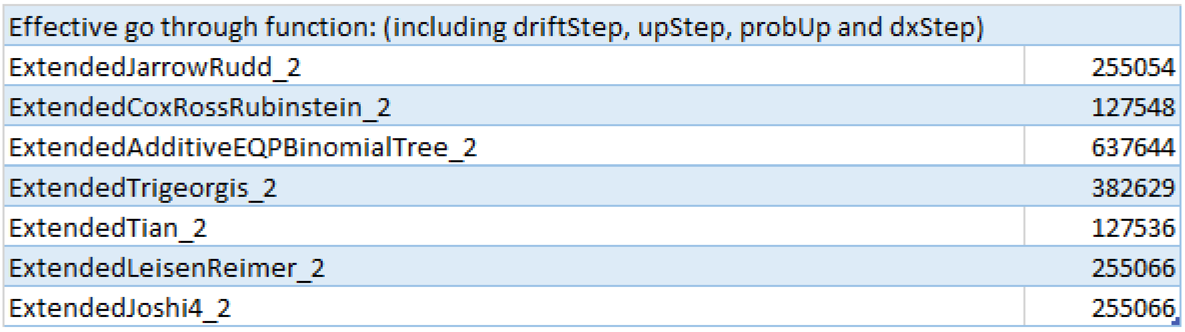
Figure 2: Current Class performances for steps = 1000



Method	European	Bermudan	American	Time (en ms)	Abs diff (en ms)	% diff
Binomial Jarrow-Rudd	3.84	4.36	4.49	392.87		
Ext Binomial Jarrow-Rudd	3.84	4.36	4.49	2304.85	1911.98	486%
Binomial Cox-Ross-Rubinstein	3.84	4.36	4.49	384.99		
Ext Binomial Cox-Ross-Rubinstein	3.84	4.36	4.49	876.57	491.58	127%
Additive equiprobabilities	3.84	4.36	4.48	390.41		
Ext Additive equiprobabilities	3.84	4.36	4.48	6592.17	6201.76	1588%
Binomial Trigeorgis	3.84	4.36	4.49	397.29		
Ext Binomial Trigeorgis	3.84	4.36	4.49	3817.39	3420.11	860%
Binomial Tian	3.84	4.36	4.49	497.05		
Ext Binomial Tian	3.84	4.36	4.49	2808.37	2311.32	465%
Binomial Leisen-Reimer	3.84	4.36	4.49	497.33		
Ext Binomial Leisen-Reimer	3.84	4.36	4.49	4764.76	4267.43	858%
Binomial Joshi	3.84	4.36	4.49	513.28		
Ext Binomial Joshi	3.84	4.36	4.49	4499.85	3986.57	776%
Time: 28737.955000 ms						

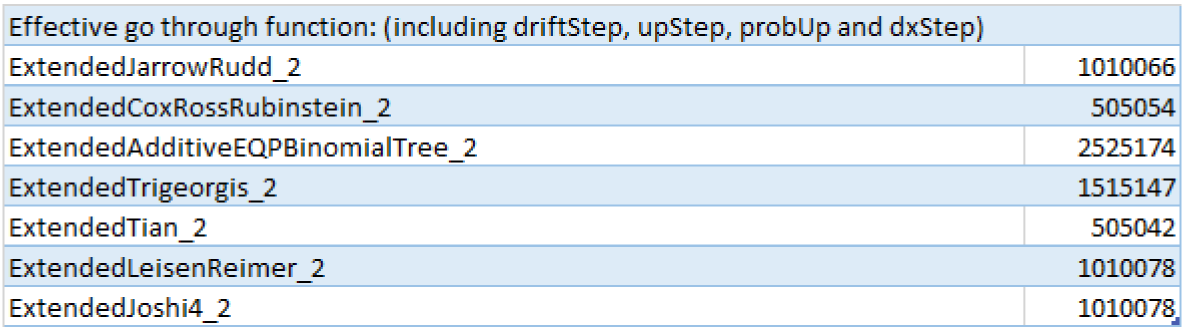
Figure 3: Current Class performances for steps = 2000

As you can see, depending on the number of time steps and the method used, the time needed may be 2 times longer up to 16 times. There is certainly a way to optimize the calculations. Let's take a look at the number of time the parameters are recalculated to see if there is redundancy :



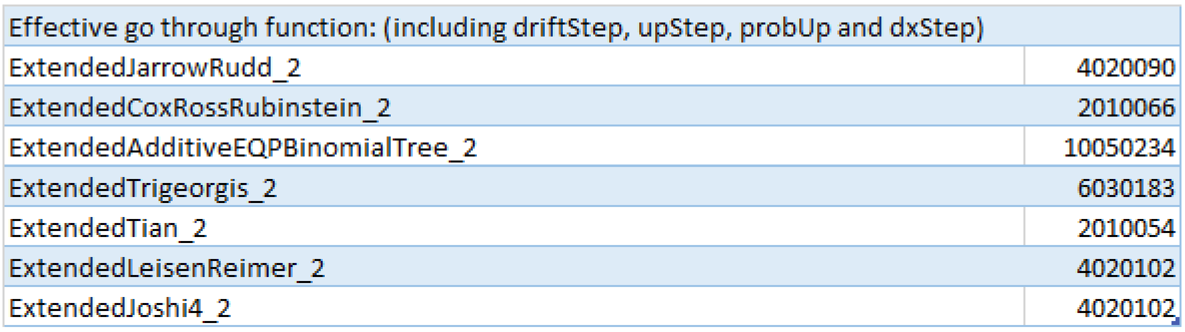
Effective go through function: (including driftStep, upStep, probUp and dxStep)	
ExtendedJarrowRudd_2	255054
ExtendedCoxRossRubinstein_2	127548
ExtendedAdditiveEQPBinomialTree_2	637644
ExtendedTrigeorgis_2	382629
ExtendedTian_2	127536
ExtendedLeisenReimer_2	255066
ExtendedJoshi4_2	255066

Figure 4: Current Class number of iterations for steps = 500



Effective go through function: (including driftStep, upStep, probUp and dxStep)	
ExtendedJarrowRudd_2	1010066
ExtendedCoxRossRubinstein_2	505054
ExtendedAdditiveEQPBinomialTree_2	2525174
ExtendedTrigeorgis_2	1515147
ExtendedTian_2	505042
ExtendedLeisenReimer_2	1010078
ExtendedJoshi4_2	1010078

Figure 5: Current Class number of iterations for steps = 1000



Effective go through function: (including driftStep, upStep, probUp and dxStep)	
ExtendedJarrowRudd_2	4020090
ExtendedCoxRossRubinstein_2	2010066
ExtendedAdditiveEQPBinomialTree_2	10050234
ExtendedTrigeorgis_2	6030183
ExtendedTian_2	2010054
ExtendedLeisenReimer_2	4020102
ExtendedJoshi4_2	4020102

Figure 6: Current Class number of iterations for steps = 2000

We can conclude from those figures that the functions driftStep, upStep, probUp and dxStep are called a way too many times than necessary : we can assume that sometimes the parameters are recalculated even if it did not change. Those functions should be called only when the values are new.

2 Solution

2.1 A Design Pattern: the Cache

Our solution is to create a cache zone for the functions that are called a lot of times : driftStep, upStep, probUp and dxStep. We found two ways to do that :

- The first solution is to create a dictionary and just initialize it in the constructor.
- The second solution is to use a design pattern. The caching design pattern is a very interesting one.

Even though the first solution is easier and faster to implement, we decided to use the second option as it is more flexible. Below is described the process of the caching design pattern.

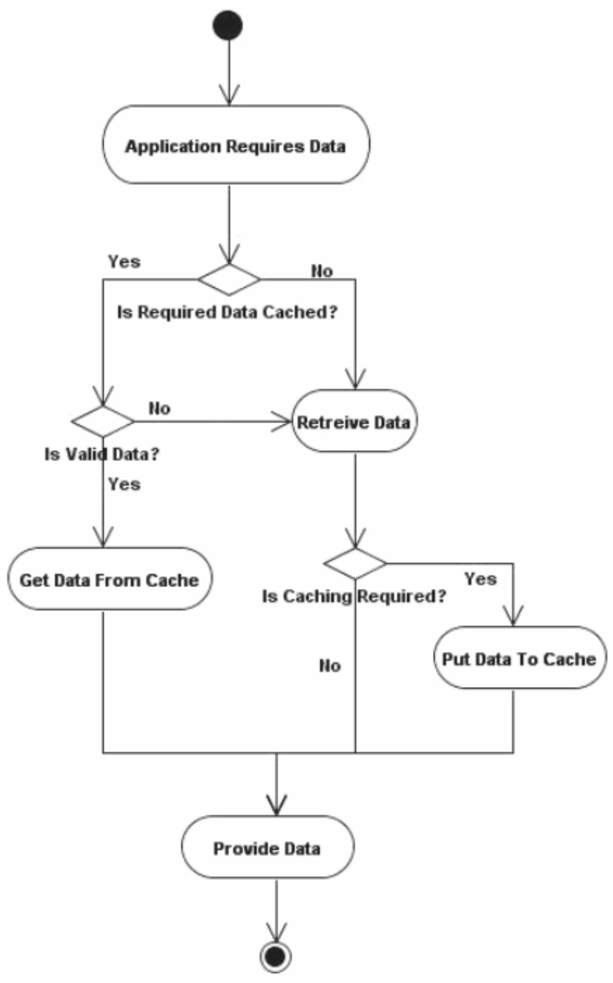


Figure 7: Caching Pattern

2.2 Implementation

We implemented the cache class and modified the ExtendedBinomialTree Class so that it uses the cache. We used several instances of the Cache class. You can find below the code of the Cache class :

```
#ifndef cache_hpp
#define cache_hpp
#include <functional>
template<class X, class Y>
class Cache
{
public:
    //CONSTRUCTOR THAT TAKE A FUNCTION
        Cache(const std::function<Y(X)> _f){
                f = _f;
        }
        //REDEFINITION OF THE () OPERATOR TO HAVE A FUNCTOR
        Y operator()(X key) const {
                auto index = value.find(key);
                if(index != value.end())
                        return value[key];
                else
                        return calculateData(key);
        }
        //TO CLEAR ALL MAP VALUE
        void clear(){
                value.clear;
        }
        //TO DELETE SPECIFIC KEY VALUE
        void erase(X key){
                value.erase(key);
        }
private:
        std::function<Y(X)> f;
        mutable std::map <X, Y> value;
        //FUNCTION TO CALCULATE VALUE
        Y calculateData(X key) const {
                put(key, f(key));
                return value[key];
        }
        //SET NEW DATA
        template<class Z>
        void put(X key, Z data) const {
                value[key] = static_cast<Y>(data);
        }
};
#endif
```

Instead of calling the specific function, it will call the cache used as a functor (i.e for exemple, this ->dxStep(t) will be changed to this->dxStepCache(t) with dxStepCache an object cache).

3 Result

3.1 Performance tests

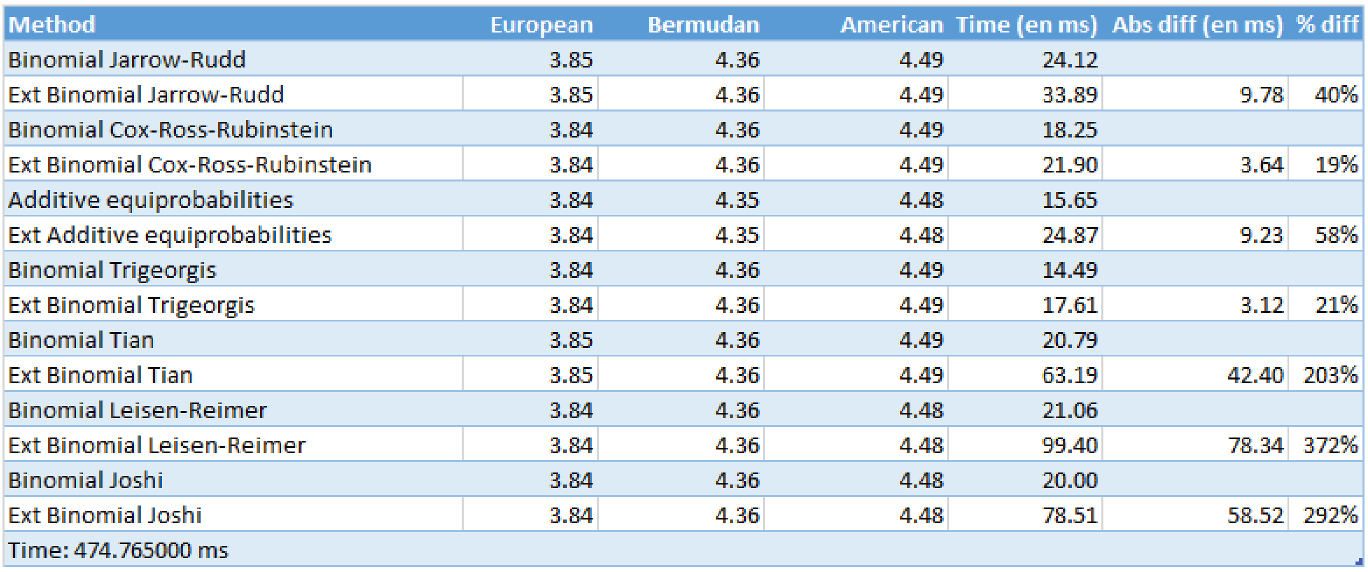
We implemented performance tests for the following types of Trees :

- Binomial Jarrow-Rudd
- Cox-Ross-Rubinstein
- Additive equiprobabilities
- Binomial Trigeorgis
- Binomial Tian
- Binomial Leisen-Reimer
- Binomial Joshi

We also tried different time steps as suggested in the subject :

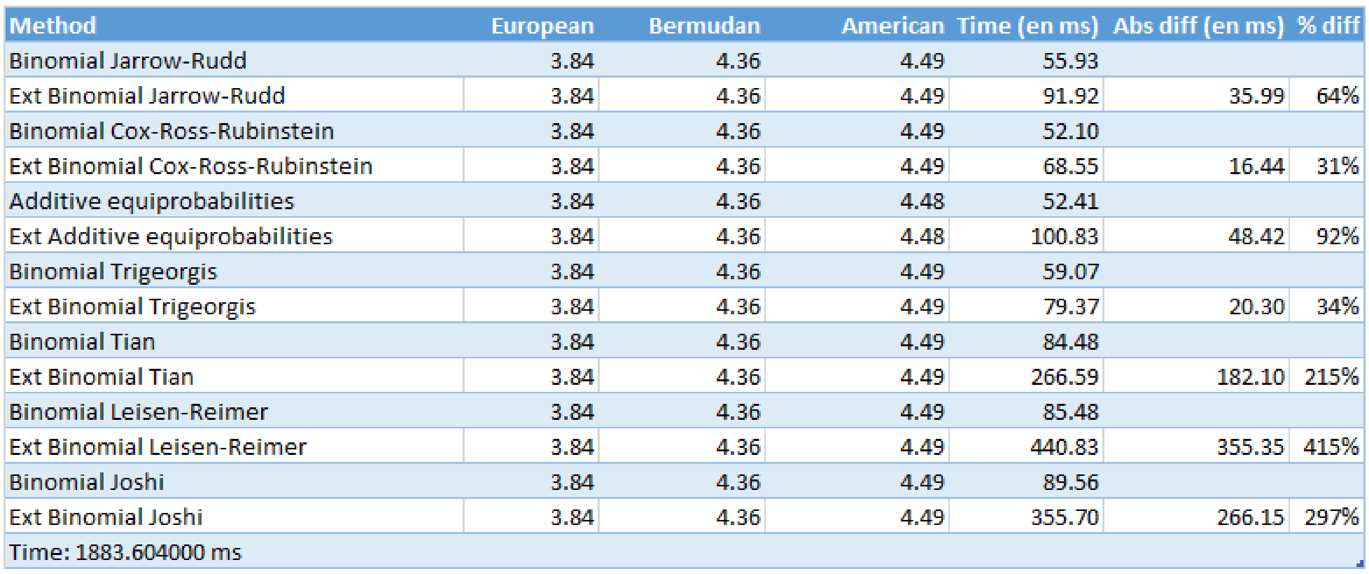
- for 500
- for 1000
- fro 2000

Let's take a look at the performances when using our solution :



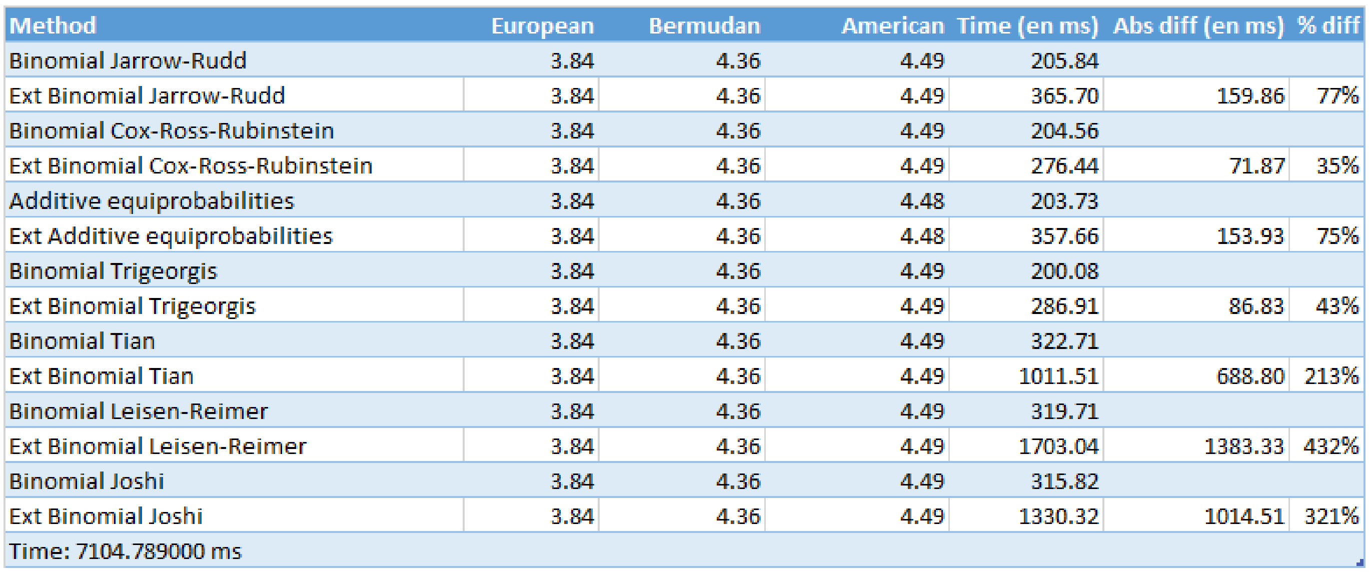
Method	European	Bermudan	American	Time (en ms)	Abs diff (en ms)	% diff
Binomial Jarrow-Rudd	3.85	4.36	4.49	24.12		
Ext Binomial Jarrow-Rudd	3.85	4.36	4.49	33.89	9.78	40%
Binomial Cox-Ross-Rubinstein	3.84	4.36	4.49	18.25		
Ext Binomial Cox-Ross-Rubinstein	3.84	4.36	4.49	21.90	3.64	19%
Additive equiprobabilities	3.84	4.35	4.48	15.65		
Ext Additive equiprobabilities	3.84	4.35	4.48	24.87	9.23	58%
Binomial Trigeorgis	3.84	4.36	4.49	14.49		
Ext Binomial Trigeorgis	3.84	4.36	4.49	17.61	3.12	21%
Binomial Tian	3.85	4.36	4.49	20.79		
Ext Binomial Tian	3.85	4.36	4.49	63.19	42.40	203%
Binomial Leisen-Reimer	3.84	4.36	4.48	21.06		
Ext Binomial Leisen-Reimer	3.84	4.36	4.48	99.40	78.34	372%
Binomial Joshi	3.84	4.36	4.48	20.00		
Ext Binomial Joshi	3.84	4.36	4.48	78.51	58.52	292%
Time: 474.765000 ms						

Figure 8: Revised Class performances for steps = 500



Method	European	Bermudan	American	Time (en ms)	Abs diff (en ms)	% diff
Binomial Jarrow-Rudd	3.84	4.36	4.49	55.93		
Ext Binomial Jarrow-Rudd	3.84	4.36	4.49	91.92	35.99	64%
Binomial Cox-Ross-Rubinstein	3.84	4.36	4.49	52.10		
Ext Binomial Cox-Ross-Rubinstein	3.84	4.36	4.49	68.55	16.44	31%
Additive equiprobabilities	3.84	4.36	4.48	52.41		
Ext Additive equiprobabilities	3.84	4.36	4.48	100.83	48.42	92%
Binomial Trigeorgis	3.84	4.36	4.49	59.07		
Ext Binomial Trigeorgis	3.84	4.36	4.49	79.37	20.30	34%
Binomial Tian	3.84	4.36	4.49	84.48		
Ext Binomial Tian	3.84	4.36	4.49	266.59	182.10	215%
Binomial Leisen-Reimer	3.84	4.36	4.49	85.48		
Ext Binomial Leisen-Reimer	3.84	4.36	4.49	440.83	355.35	415%
Binomial Joshi	3.84	4.36	4.49	89.56		
Ext Binomial Joshi	3.84	4.36	4.49	355.70	266.15	297%
Time: 1883.604000 ms						

Figure 9: Revised Class performances steps = 1000



Method	European	Bermudan	American	Time (en ms)	Abs diff (en ms)	% diff
Binomial Jarrow-Rudd	3.84	4.36	4.49	205.84		
Ext Binomial Jarrow-Rudd	3.84	4.36	4.49	365.70	159.86	77%
Binomial Cox-Ross-Rubinstein	3.84	4.36	4.49	204.56		
Ext Binomial Cox-Ross-Rubinstein	3.84	4.36	4.49	276.44	71.87	35%
Additive equiprobabilities	3.84	4.36	4.48	203.73		
Ext Additive equiprobabilities	3.84	4.36	4.48	357.66	153.93	75%
Binomial Trigeorgis	3.84	4.36	4.49	200.08		
Ext Binomial Trigeorgis	3.84	4.36	4.49	286.91	86.83	43%
Binomial Tian	3.84	4.36	4.49	322.71		
Ext Binomial Tian	3.84	4.36	4.49	1011.51	688.80	213%
Binomial Leisen-Reimer	3.84	4.36	4.49	319.71		
Ext Binomial Leisen-Reimer	3.84	4.36	4.49	1703.04	1383.33	432%
Binomial Joshi	3.84	4.36	4.49	315.82		
Ext Binomial Joshi	3.84	4.36	4.49	1330.32	1014.51	321%
Time: 7104.789000 ms						

Figure 10: Revised Class performances for steps = 2000

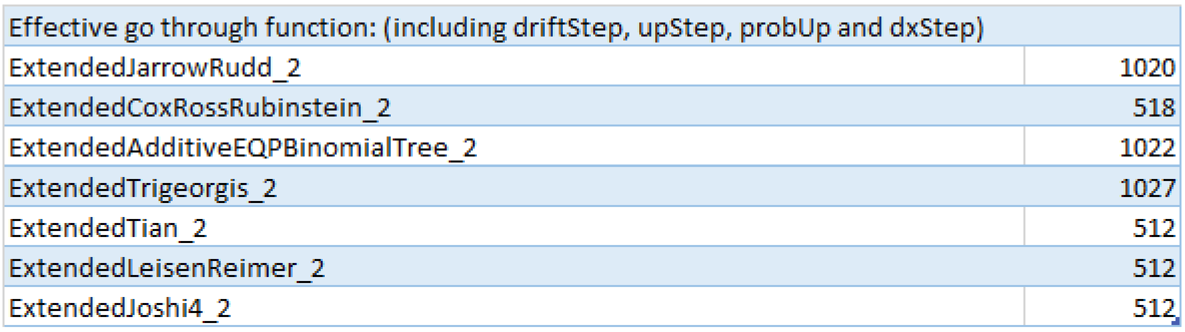
If you compare the table on Figure 8, 9 and 10 with respectively Figure 1, 2 and 3, you can clearly notice that the revised class is way more faster in an overall situation:

- for 500: it went from 1.8s to 0.5s which is almost 4 times quicker
- for 1000: it went from 8.5s to 1.8s which is almost 5 times quicker

- for 2000: it went from 29s to 7s which is almost 4 times quicker

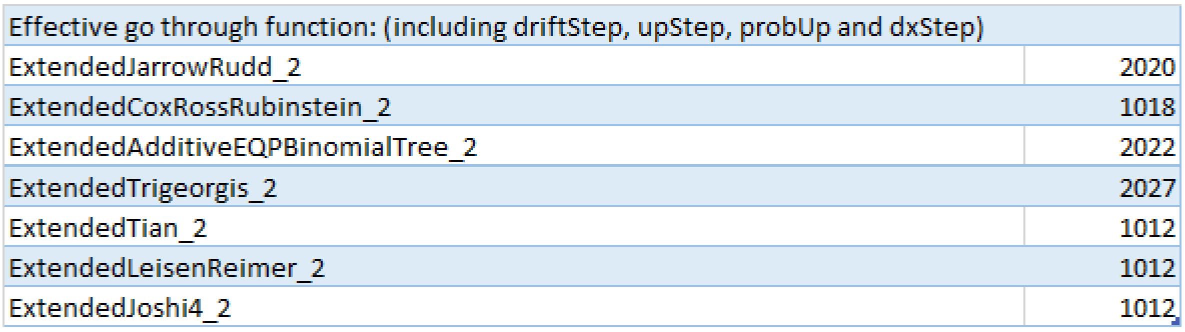
The performances are now a way better. The time is divided by at least 4 for the different time steps.

Let's now check if the four functions quoted before are less called :



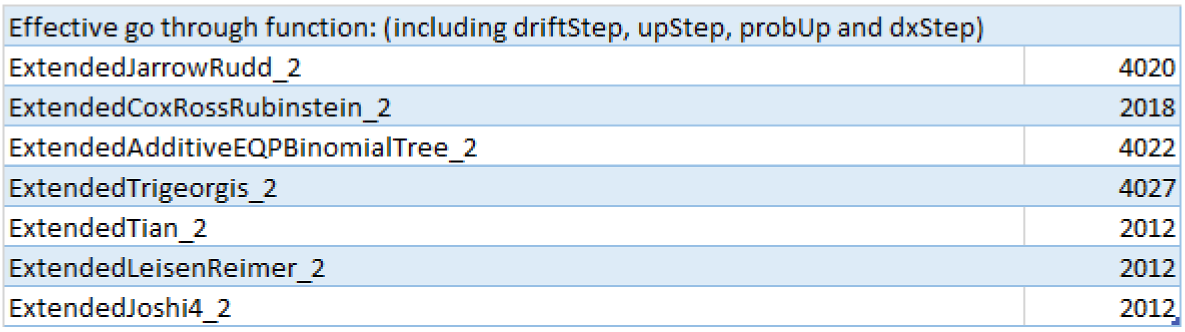
Effective go through function: (including driftStep, upStep, probUp and dxStep)	
ExtendedJarrowRudd_2	1020
ExtendedCoxRossRubinstein_2	518
ExtendedAdditiveEQPBinomialTree_2	1022
ExtendedTrigeorgis_2	1027
ExtendedTian_2	512
ExtendedLeisenReimer_2	512
ExtendedJoshi4_2	512

Figure 11: Revised Class number of iteration for steps = 500



Effective go through function: (including driftStep, upStep, probUp and dxStep)	
ExtendedJarrowRudd_2	2020
ExtendedCoxRossRubinstein_2	1018
ExtendedAdditiveEQPBinomialTree_2	2022
ExtendedTrigeorgis_2	2027
ExtendedTian_2	1012
ExtendedLeisenReimer_2	1012
ExtendedJoshi4_2	1012

Figure 12: Revised Class number of iteration for steps = 1000



Effective go through function: (including driftStep, upStep, probUp and dxStep)	
ExtendedJarrowRudd_2	4020
ExtendedCoxRossRubinstein_2	2018
ExtendedAdditiveEQPBinomialTree_2	4022
ExtendedTrigeorgis_2	4027
ExtendedTian_2	2012
ExtendedLeisenReimer_2	2012
ExtendedJoshi4_2	2012

Figure 13: Revised Class number of iteration for steps = 2000

Indeed, we hugely reduced the number of times the functions are called when using the cache. The datas are now stored and the functions are called only when necessary.

3.2 Reduction of the Complexity

If you look at figure 4, 5 and 6 you can see that the number of iterations in the 4 functions go from $\frac{N^2}{4}$ up to N. But those iterations concern more than one function. However, we can say that the previous approximate complexity was O(N^2).

Now, if you look at figure 11, 12 and 13, the number of iterations are kind of similar to the number of time steps (multiply by one or 2 depending on the number of functions used in the method). The complexity is now approximately O(N)

Using a cache pattern allow us to drastically reduce the complexity of our class which is a huge benefit for the execution time.