

[Help](#)

```

#if defined(PremiaCurrentVersion) && PremiaCurrentVersion < (2007+2) //The "#els
#else

#include <cmath>
#include "generator.h"

#include <vector>

#ifndef montecarlo2_h_
#define montecarlo2_h_

//Euler Scheme
//X(p)=X(p-1)+b(X(p), ph)h+sigma(X(p), ph)*sqrt(h)*random variable
//The functions b(), sigma() are differentes for differentes models
//so we give the pointer to a model where these functions are described

//Also the vector of random variable can be different (with component correlated)
//so we give the pointer to a random vector.

//_nstep - number of partitions
std::vector<double> scheme_euler(rv_vector *_ptr_rv, model *_ptr_model, int _nst
{
    //h- step of discretisation
    double h = _ptr_model->T / _nstep;

    std::vector<double> x = _ptr_model->x0;

    for (int i = 0; i < _nstep; i++)
    {
        x = x + _ptr_model->f_b(x, ((double)i) * h) * h + _ptr_model->f_sigma(x, (
    }
    return x;
}

//Ninomiya-Victoir Scheme (Kusuoka Scheme)
//The functions f_1, f_2 and the vector of random variable are differentes for d

//_nstep - number of partitions

```

```

std::vector<double> scheme_kusuoka(rv_vector *_ptr_rv, model *_ptr_model, int _n
{
    //h- step of discretisation
    double h = _ptr_model->T / _nstep;

    std::vector<double> x = _ptr_model->x0;

    rv_bernoulli rv_b(0.5, -1, 1, generator);

    double rv_b_real;

    for (int i = 0; i < _nstep; i++)
    {
        x = _ptr_model->exp_V0(0.5 * h, x);

        rv_b_real = rv_b.get_rv();
        std::vector<double> y = _ptr_rv->get_rv();

        if (rv_b_real == 1)
            x = _ptr_model->f_1(x, h, y);
        else
            x = _ptr_model->f_2(x, h, y);

        x = _ptr_model->exp_V0(0.5 * h, x);
    }

    return x;
}

//the Monte Carlo Method
//we estimate the solution of stochastic differential equation by discretisation
//we calculate a function of this solution (x)
//then (the last step) we apply the Monte Carlo Method to x:
//we construct the sum of x (_niter times) (=nsum_x)

//_nstep step of discretisation (for discretisation scheme)
//_niter number of trajectories in the Monte Carlo Method
//_ptr_rv pointer to the vector of random variable (for discretisation scheme)
//_function function of processus discretized
//_function_delta function of option delta

```

```

//_schema name of the scheme
//_ptr_model pointer to a model
//_nerror Monte Carlo Method error (_nerror*_nerror is a a variance of Monte Carlo)
//_ndelta delta of option
//_nerror_delta delta error

template<class A, class B, class C> double monte_carlo
(int _nstep, int _niter, rv_vector *_ptr_rv, A _function, B _function_delta, C _generator)
{
    std::vector<double> nres = _ptr_model->x0;

    double x = 0.;
    double nsum_x = 0.;
    double nsum_x_x = 0.;

    double ndelta = 0.;
    double nsum_delta = 0.;
    double nsum_delta_delta = 0.;

    double one_n = (1. / ((double)_niter));

    for (int i = 0; i < _niter; i++)
    {
        nres = _scheme(_ptr_rv, _ptr_model, _nstep, generator);
        x = _function(nres, _ptr_model);
        nsum_x += x;
        nsum_x_x += x * x;

        ndelta = _function_delta(nres, _ptr_model);
        nsum_delta += ndelta;
        nsum_delta_delta += ndelta * ndelta;
    }

    _nerror = sqrt(one_n * std::abs(nsum_x_x - one_n * nsum_x * nsum_x) / ((double)_niter));

    _ndelta = nsum_delta * one_n;
    _nerror_delta = sqrt(one_n * std::abs(nsum_delta_delta - one_n * nsum_delta * nsum_delta) / ((double)_niter));

    return nsum_x * one_n;
}

```

```
}
```

```
//the Monte Carlo Method / Varince reduction
```

```
//we estimate the solution of stochastic differential equation by discretisation
```

```
//we calculate a function of this solution (x)
```

```
//we construct a control variable (y)
```

```
//then we apply the varinace reduction technique
```

```
//In place of sum_x/_niter we estimate (nsum_x -alpha*nsum_y)/_niter+alpha*nesp_
```

```
//this expression gives a mean empiric with a smaller variance.
```

```
//where alpha is calculcd by (covariance(x,y)/variance(y))
```

```
//here x-variable estimed and y - control variable ;
```

```
//the parameters are the same that one for monte_carlo
```

```
//_ncorr the correlation coefficient between variable estimated and control vari
```

```
template<class A, class B, class C> double monte_carlo2
```

```
(int _nstep, int _niter, rv_vector *_ptr_rv, A _function, B _function_delta, C _
```

```
{
```

```
    double epsilon = DBL_EPSILON;
```

```
    std::vector<double> nres = _ptr_model->x0;
```

```
    double x = 0.;
```

```
    double y = 0.;
```

```
    double nsum_x = 0.;
```

```
    double nsum_y = 0.;
```

```
    double nsum_x_y = 0.;
```

```
    double nsum_y_y = 0.;
```

```
    double nsum_x_x = 0.;
```

```
    double ndelta = 0.;
```

```
    double nsum_delta = 0.;
```

```
    double nsum_delta_delta = 0.;
```

```
    //mean of a control variable
```

```
    double nvar = 0.;
```

```

double nesp_y = _ptr_model->f_esp(nvar);

double one_n = (1. / ((double)_niter));

for (int i = 0; i < _niter; i++)
{
    nres = _scheme(_ptr_rv, _ptr_model, _nstep, generator);

    x = _function(nres, _ptr_model);
    y = _ptr_model->f_control(nres);

    nsum_x += x;
    nsum_y += y;
    nsum_x_y += x * y;
    nsum_y_y += y * y;
    nsum_x_x += x * x;

    ndelta = _function_delta(nres, _ptr_model);
    nsum_delta += ndelta;
    nsum_delta_delta += ndelta * ndelta;
}

double ncov_x_y = one_n * nsum_x_y - one_n * one_n * nsum_y * nsum_x;
double nvar_x = one_n * nsum_x_x - one_n * one_n * nsum_x * nsum_x;
double nvar_y = one_n * nsum_y_y - one_n * one_n * nsum_y * nsum_y;

double alpha = 0.;

alpha = (std::abs(nvar_y) <= epsilon) ? 0. : ncov_x_y / nvar_y;

_nerror = (std::abs(nvar_y) <= epsilon) ? sqrt(one_n * std::abs(nvar_x)) : sqrt(nvar_x);

_ncorr = ((std::abs(nvar_y) <= epsilon) || (std::abs(nvar_x) <= epsilon)) ? 0. : ncov_x_y / (nvar_x * nvar_y);

_ndelta = nsum_delta * one_n;
_nerror_delta = sqrt(one_n * std::abs(nsum_delta_delta - one_n * nsum_delta * nsum_delta));

return one_n * nsum_x - alpha * one_n * nsum_y + alpha * nesp_y;
}

```

```
#endif  
#endif //PremiaCurrentVersion
```