

Help

```

#include <iostream>
#include <cmath>
#include <cstdlib>
#include <cstring>

#include "math/ImportanceSampling_jl/src/Model.hpp"
#include "math/ImportanceSampling_jl/src/Option.hpp"
#include "math/ImportanceSampling_jl/src/PremiaOption.hpp"
#include "math/ImportanceSampling_jl/src/parser.hpp"
#include "math/ImportanceSampling_jl/src/MonteCarlo.hpp"
#include "math/ImportanceSampling_jl/src/MonteCarloMode.hpp"
#include "math/ImportanceSampling_jl/src/MultiLevelMonteCarlo.hpp"

using namespace std;

/**
 * Run the right MonteCarlo routine
 *
 * @param rng PnlRng
 * @param map an instance of Param
 * @param delta a boolean to decide whether to compute the delta
 * @param sample_average a boolean to decide whether to use IS with sample
 * averaging.
 * @param price a boolean to tell to only compute the price
 * @param couple a boolean to tell to couple RM and MC
 * @param average a boolean to tell to use an averaging RM method
 * @param poisson_only a boolean to decide to only apply the importance
 * sampling to the Poisson part
 * @param poisson a boolean to decide to apply the importance
 * sampling both to the Poisson part and the Brownian part
 * @param reduced a boolean telling to use a reduced importance sampling
 * parameter
 */
void MonteCarloWrapper(PnlRng *rng, const Param &map, bool price, bool delta,
                      bool sample_average, bool couple, bool average, bool pois
                      bool poisson, double &prix, double &std_dev)
{
    PnlVect *drift, *dprice, *dpricestd_dev;

```

```
bool reduced = true;

drift = pnl_vect_new();
dprice = pnl_vect_new();
dpricestd_dev = pnl_vect_new();

MonteCarloBase *mc = CreateMonteCarlo(rng, map, reduced);

if (price)
{
    if (delta)
    {
        mc->PriceDelta(prix, std_dev, dprice, dpricestd_dev);
    }
    else
    {
        mc->Price(prix, std_dev);
    }
}

else if (average && !couple)
{
    int nb_proj;
    mc->RM_routine_average(NULL, NULL, drift, prix, std_dev, nb_proj, 2);
}

else if (average && couple)
{
    int nb_proj;
    mc->RM_routine_average_coupled(NULL, NULL, drift, prix, std_dev, nb_proj,
}

else if (sample_average)
{
    if (delta)
    {
        mc->mc_sample_averaging_delta(drift, prix, std_dev, dprice, dpricestd_dev);
    }
    else
    {
        PnlVect *mu = pnl_vect_new();
        if (poisson_only)
            mc->mc_sample_averaging_poisson(mu, prix, std_dev, true);
    }
}
```

```

        else if (poisson)
            mc->mc_sample_averaging_gaussian_poisson(drift, mu, prix, std_dev, t);
        else
            mc->mc_sample_averaging(drift, prix, std_dev, true);
        pnl_vect_free(&mu);
    }
}
else if (!couple)
{
    int nb_proj;
    mc->RM_routine(NULL, drift, prix, std_dev, nb_proj);
}
else
{
    int nb_proj;
    mc->RM_routine_coupled(NULL, drift, prix, std_dev, nb_proj);
}

delete mc;
pnl_vect_free(&drift);
pnl_vect_free(&dprice);
pnl_vect_free(&dpricestd_dev);
}

void MultiLevelMonteCarloWrapper(PnlRng *rng, const Param &map, bool use_IS, double &price)
{
    MultiLevelMonteCarlo<MonteCarloModeReduced, MonteCarloModeReducedMultiLevel>
    mc(use_IS)
    {
        PnlVect *drift = pnl_vect_new();
        mc.mc_sample_averaging(drift, price, std_dev, false);
        pnl_vect_free(&drift);
    }
    else
        mc.Price(price, std_dev);
}

extern "C" {

```

```
#include "optype.h"
```

```
PnlVect* ComputeEuropeanBSDrift(PnlVect *spot, PnlVect *sig, PnlVect *divid, dou
{
```

```
    int size = spot->size;
    Param P;
    std::vector<double> sig_v(sig->array, sig->array + size);
    std::vector<double> spot_v(spot->array, spot->array + size);
    std::vector<double> divid_v(divid->array, divid->array + size);
```

```
    P.insert("option type", T_STRING, string("PremiaOption"));
    P.insert("model type", T_STRING, string("bs"));
    P.insert("option size", T_INT, size);
    P.insert("spot", T_VECTOR, spot_v);
    P.insert("maturity", T_DOUBLE, maturity);
    P.insert("volatility", T_VECTOR, sig_v);
    P.insert("interest rate", T_DOUBLE, r);
    P.insert("dividend rate", T_VECTOR, divid_v);
    P.insert("correlation", T_DOUBLE, rho);
    P.insert("timestep number", T_INT, 1);
    P.insert("sample number", T_LONG, size_t(n_samples));
    P.insert("payoff numfunc", T_PTR, static_cast<void *>(p_numfunc));
```

```
    bool reduced = true;
    MonteCarloBase *mc = CreateMonteCarlo(rng, P, reduced);
    PnlVect *drift = pnl_vect_new();
    mc->sample_averaging_newton(drift, false);
    delete mc;
    return drift;
```

```
}
```

```
}
```