

[Help](#)

```
#include <iostream>
#include <iostream>
#include <cstdlib>
#include <cmath>

#include "
href../../../../common/math/ImportanceSampling_jl/src/Model_h_src.pdfmath/Import
#include "pnl/pnl_matrix.h"
#include "
href../../../../common/math/jlparser/include/jlparser/parser_h_src.pdfjlparser/p
#include "
href../../../../common/math/ImportanceSampling_jl/src/BlackScholesModel_h_src.pdf
#include "
href../../../../common/math/ImportanceSampling_jl/src/MertonModel_h_src.pdfmath/
#include "
href../../../../common/math/ImportanceSampling_jl/src/KouModel_h_src.pdfmath/Imp
#include "
href../../../../common/math/ImportanceSampling_jl/src/BNSModel_h_src.pdfmath/Imp
#include "
href../../../../common/math/ImportanceSampling_jl/src/DupireModel_h_src.pdfmath/
#include "
href../../../../common/math/ImportanceSampling_jl/src/SABRModel_h_src.pdfmath/Im
#include "
href../../../../common/math/ImportanceSampling_jl/src/HestonModel_h_src.pdfmath/

using namespace std;

BaseModel *instantiate_model(const Param &P)
{
    BaseModel *mod = NULL;
    string modelType;
    P.extract("model type", modelType);

    if (modelType == "bs")
        mod = new BlackScholesModel(P);
    else if (modelType == "local_vol")
        mod = new DupireModel(P);
    else if (modelType == "merton")
        mod = new MertonModel(P);
```

```

else if (modelType == "kou")
    mod = new KouModel(P);
else if (modelType == "bns")
    mod = new BNSModel(P);
else if (modelType == "sabr")
    mod = new SABRModel(P);
else if (modelType == "heston")
    mod = new HestonModel(P);
else
{
    cout << "BaseModel " << modelType << " unknow. Abort." << endl;
    abort();
}

return mod;
}

BaseModel::BaseModel()
{
    nTimeSteps = 1;
    workVector = NULL;
    workMatrix = NULL;
    covChol = NULL;
    pathMatrix = NULL;
    Gincr = NULL;
    Gincr_drift = NULL;
}

void BaseModel::GenericConstructor(const Param &P)
{
    nTimeSteps = 1;
    rho = 1.;
    P.extract("maturity", maturity);
    P.extract("timestep number", nTimeSteps, true);
    P.extract("spot", init, size);
    P.extract("interest rate", interest);
    P.extract("dividend rate", dividend, size, true);
    if (dividend == NULL)
    {
        dividend = pnl_vect_create_from_zero(size);
    }
}

```

```

if (!P.extract("correlation", rho, true) && (size != 1))
{
    perror("correlation is missing");
    exit(1);
}
sqrt_nTimeSteps = sqrt(double(nTimeSteps));
dt = maturity / double(nTimeSteps);
sqrt_dt = sqrt(dt);
covChol = pnl_mat_create(brownianSize, brownianSize);
pathMatrix = pnl_mat_create(nTimeSteps + 1, size);
Gincr = pnl_mat_create(nTimeSteps, brownianSize);
Gincr_drift = pnl_mat_create(nTimeSteps, brownianSize);

if (computeCovChol() == FAIL)
{
    perror("Correlation matrix is not positive definite");
    exit(1);
}

workVector = pnl_vect_new();
workMatrix = pnl_mat_new();
}

BaseModel::BaseModel(const Param &P)
{
    P.extract("option size", size);
    brownianSize = size;
    GenericConstructor(P);
}

BaseModel::~BaseModel()
{
    pnl_mat_free(&covChol);
    pnl_mat_free(&pathMatrix);
    pnl_mat_free(&Gincr);
    pnl_mat_free(&Gincr_drift);
    pnl_vect_free(&init);
    pnl_vect_free(&dividend);
    pnl_vect_free(&workVector);
    pnl_mat_free(&workMatrix);
}

```

```

}

/* set the value of the correlation parameter and update the
 * covariance matrix and its derivative */
int BaseModel::computeCovChol()
{
    int i, j;
    for (i = 0 ; i < size ; i++)
    {
        MLET(covChol, i, i) = 1.0;
        for (j = 0 ; j < i ; j++)
        {
            MLET(covChol, i, j) = rho;
            MLET(covChol, j, i) = rho;
        }
    }
    return pnl_mat_chol(covChol);
}

double BaseModel::Rho() const
{
    return this->rho;
}

int BaseModel::Size() const
{
    return this->size;
}

void BaseModel::print() const
{
    cout << " interest rate : " << interest << endl;
    cout << " dividend rate : "; pnl_vect_print_asrow(dividend);
    if (size > 1) { cout << " correlation " << rho << endl; }
    cout << " Number of nTimeStepss : " << nTimeSteps << endl;
    cout << " Spot : "; pnl_vect_print_asrow(init);
}

void BaseModel::path(const PnlVect *drift)
{
    pnl_mat_clone(Gincr_drift, Gincr);
}

```

```

    for (int j = 0 ; j < brownianSize ; j++)
    {
        const double drift_j = sqrt_dt * GET(drift, j);
        for (int i = 0 ; i < nTimeSteps ; i++)
        {
            MLET(Gincr_drift, i, j) += drift_j;
        }
    }
    path();
}

void BaseModel::path(PnlRng *rng)
{
    pnl_mat_rng_normal(Gincr, nTimeSteps, brownianSize, rng);
    pnl_mat_clone(Gincr_drift, Gincr);
    path();
}

void BaseModel::path(PnlRng *rng, const PnlVect *drift)
{
    pnl_mat_rng_normal(Gincr, nTimeSteps, brownianSize, rng);
    path(drift);
}

void BaseModel::pathFull(PnlRng *rng, const PnlVect *drift)
{
    pnl_mat_rng_normal(Gincr, nTimeSteps, brownianSize, rng);
    pathFull(drift);
}

void BaseModel::pathFull(const PnlVect *drift)
{
    pnl_mat_set_from_ptr(Gincr_drift, drift->array);
    pnl_mat_plus_mat(Gincr_drift, Gincr);
    path();
}

void BaseModel::shiftPath(int d, double h)
{
    for (int i = 0 ; i <= nTimeSteps ; i++)
    {

```

```

        MLET(pathMatrix, i, d) *= (1 + h);
    }
}

void BaseModel::unshiftPath(int d, double h)
{
    for (int i = 0 ; i <= nTimeSteps ; i++)
    {
        MLET(pathMatrix, i, d) /= (1 + h);
    }
}

//
// Stochastic Volatility class
//

StocVolModel::StocVolModel() : BaseModel(), sigma0(NULL), voVol(NULL), sigmaVect

StocVolModel::StocVolModel(const Param&P)
{
    P.extract("option size", size);
    P.extract("initial volatility", sigma0, size);
    P.extract("volatility of volatility", voVol, size);
    P.extract("asset vol correlation", gamma);
    brownianSize = 2 * size;
    GenericConstructor(P);
    sigmaVector = pnl_vect_new();
}

StocVolModel::~StocVolModel()
{
    pnl_vect_free(&voVol);
    pnl_vect_free(&sigma0);
    pnl_vect_free(&sigmaVector);
}

int StocVolModel::computeCovChol()
{
    const double cross_corel = gamma * rho;
    const double correl_W = gamma * gamma;
    // Compute the overall correlation matrix

```

```

// First diagonal block
for (int i = 0; i < size; i++)
{
    for (int j = 0; j < i ; j++)
    {
        MLET(covChol, i, j) = rho;
        MLET(covChol, j, i) = rho;
    }
}
// Second diagonal block
for (int i = size; i < brownianSize; i++)
{
    for (int j = size; j < i; j++)
    {
        MLET(covChol, i, j) = correl_W;
        MLET(covChol, j, i) = correl_W;
    }
}
// Extra diagonal block
for (int i = size; i < brownianSize; i++)
{
    for (int j = 0; j < size; j++)
    {
        MLET(covChol, i, j) = cross_corel;
        MLET(covChol, j, i) = cross_corel;
    }
}
// Set the block diagonal terms
for (int i = 0; i < size; i++)
{
    MLET(covChol, i, i) = 1.;
    MLET(covChol, i + size, i + size) = 1.;
    MLET(covChol, i, i + size) = gamma;
    MLET(covChol, i + size, i) = gamma;
}
return pnl_mat_chol(covChol);
}

```

```

//
// JumpModel class
//

JumpModel::JumpModel() : BaseModel() { }

JumpModel::~JumpModel()
{
    if (sigma) pnl_vect_free(&sigma);
    if (lambda) pnl_vect_free(&lambda);
    if (lambdaFull) pnl_vect_free(&lambdaFull);
    if (levyDrift) pnl_vect_free(&levyDrift);
    pnl_mat_free(&jumpsMat);
    pnl_mat_free(&poissonMat);
}

JumpModel::JumpModel(const Param &P)
    : BaseModel(P)
{
    poissonSize = size;
    if (P.extract("poisson size", poissonSize, true))
    {
        if ((size == 1 && poissonSize != 1) ||
            (size > 1 && ((poissonSize != size) &&
                          (poissonSize != (size + 1)) &&
                          (poissonSize != 1))))
        {
            printf("poissonMat size is wrong\ n");
            abort();
        }
    }
    P.extract("volatility", sigma, size);
    P.extract("jump intensity", lambda, poissonSize);
    poissonMat = pnl_mat_create(nTimeSteps, poissonSize);
    jumpsMat = pnl_mat_create(nTimeSteps, poissonSize);
    lambdaFull = pnl_vect_create(poissonSize * nTimeSteps);
    levyDrift = pnl_vect_create(size);
    for (int i = 0 ; i < nTimeSteps ; i++)
    {
        PnlVect part = pnl_vect_wrap_subvect(lambdaFull, i * poissonSize, poissonS
        pnl_vect_clone(&part, lambda);
    }
}

```



```

    }
    pnl_vect_mult_double(lambdaFull, dt);
}

/**
 * Compute one path of a generic jump model of the form
 * BlackScholes x jumpsMat
 */
void JumpModel::path()
{
    // Time 0
    pnl_mat_set_row(pathMatrix, init, 0);

    for (int j = 1 ; j <= nTimeSteps ; j++)
    {
        PnlVect G_j = pnl_vect_wrap_mat_row(Gincr_drift, j - 1);
        pnl_mat_mult_vect_inplace(workVector, covChol, &G_j);
        for (int i = 0 ; i < size ; i++)
        {
            int i_poisson = (poissonSize == 1 ? 0 : i);
            double sigma_i = GET(sigma, i);
            MLET(pathMatrix, j, i) = MGET(pathMatrix, j - 1, i) *
                                   MGET(jumpsMat, j - 1, i_poisson) *
                                   exp(GET(levyDrift, i) * dt
                                       + sqrt_dt * sigma_i * GET(workVector, i));
            if (poissonSize == size + 1)
            {
                MLET(pathMatrix, j, i) *= MGET(jumpsMat, j - 1, size);
            }
        }
    }
}

void JumpModel::path(PnlRng *rng)
{
    pnl_mat_rng_normal(Gincr, nTimeSteps, brownianSize, rng);
    pnl_mat_clone(Gincr_drift, Gincr);
    pathMu_aux(rng, lambda);
}

void JumpModel::path(PnlRng *rng, const PnlVect *drift)

```

```

{
    pathMuTheta(rng, drift, lambda);
}

void JumpModel::pathFull(PnlRng *rng, const PnlVect *drift)
{
    pnl_mat_rng_normal(Gincr, nTimeSteps, brownianSize, rng);
    pnl_mat_set_from_ptr(Gincr_drift, drift->array);
    pnl_mat_plus_mat(Gincr_drift, Gincr);
    pathMu_aux(rng, lambda);
}

void JumpModel::pathMuFull(PnlRng *rng, const PnlVect *mu)
{
    pnl_mat_rng_normal(Gincr, nTimeSteps, brownianSize, rng);
    pnl_mat_clone(Gincr_drift, Gincr);
    pathMuFull_aux(rng, mu);
}

void JumpModel::pathMuThetaFull(PnlRng *rng, const PnlVect *drift, const PnlVect *mu)
{
    pnl_mat_rng_normal(Gincr, nTimeSteps, brownianSize, rng);
    pnl_mat_set_from_ptr(Gincr_drift, drift->array);
    pnl_mat_plus_mat(Gincr_drift, Gincr);
    pathMuFull_aux(rng, mu);
}

void JumpModel::pathMu(PnlRng *rng, const PnlVect *mu)
{
    pnl_mat_rng_normal(Gincr, nTimeSteps, brownianSize, rng);
    pnl_mat_clone(Gincr_drift, Gincr);
    pathMu_aux(rng, mu);
}

void JumpModel::pathMuTheta(PnlRng *rng, const PnlVect *drift, const PnlVect *mu)
{
    pnl_mat_rng_normal(Gincr, nTimeSteps, brownianSize, rng);
    for (int i = 0 ; i < nTimeSteps ; i++)
    {
        for (int j = 0 ; j < brownianSize ; j++)
        {

```

```

        MLET(Gincr_drift, i, j) = MGET(Gincr, i, j) + sqrt_dt * GET(drift, j);
    }
}
pathMu_aux(rng, mu);
}

void JumpModel::print() const
{
    cout << " poisson size : " << poissonSize << endl;
    cout << " sigma : ";
    pnl_vect_print_asrow(sigma);
    cout << " lambda : ";
    pnl_vect_print_asrow(lambda);
    BaseModel::print();
}

```