

[Help](#)

```

#include <stdlib.h>
#include "pnl/pnl_complex.h"
#include "pnl/pnl_vector.h"
#include "pnl/pnl_random.h"
#include "hes1d_std.h"
#include "enums.h"

#if defined(PremiaCurrentVersion) && PremiaCurrentVersion < (2011+2) //The "#els
static int CHK_OPT(MC_GlassermanKim_Heston)(void *Opt, void *Mod)
{
    return NONACTIVE;
}
int CALC(MC_GlassermanKim_Heston)(void *Opt, void *Mod, PricingMethod *Met)
{
    return AVAILABLE_IN_FULL_PREMIA;
}
#else

//-----Sample gthe law X1 by trancation series.
static double X_1_sample(double order_tr, double t, double kappa, double sigma,
{
    //-----Declaration of variable
    double lambda_n;
    double gamma_n;
    int pss;
    int j, n;
    double tmp;

    //-----Compte the sum part

    tmp = 0.;
    for (n = 1; n <= order_tr; n++)
    {
        lambda_n = 16.*M_PI * M_PI * ((double)n) * ((double)n) / (sigma * sigma *
        gamma_n = (kappa * kappa * t * t + 4.*M_PI * M_PI * ((double)n) * ((double)

        pss = pnl_rand_poisson(lambda_n * (v0 + vt), generator);

```

```

        for (j = 1; j <= pss; j++)
            tmp = tmp + pnl_rand_exp(1., generator) / gamma_n;
    }

    //-----compute the rest

    lambda_n = 6.*(v0 + vt) * ((double)order_tr) / (sigma * sigma * t);
    gamma_n = sigma * sigma * t * t / (3.*M_PI * M_PI * ((double)order_tr) * ((double)order_tr));

    tmp = tmp + pnl_rand_gamma(lambda_n, gamma_n, generator);

    //printf("The value of tmp = %f \ n",tmp);
    return tmp;
}

//-----Sample gthe law X1 by trancation series.
static double X_2_sample(double order_tr, double t, double kappa, double sigma,
{
    //-----Declaration of variable
    double lambda_n, moy, var;
    double gamma_n;
    double theta_p;

    int n;
    double tmp;

    //-----Compute the sum part

    tmp = 0.;
    for (n = 1; n <= order_tr; n++)
    {
        gamma_n = (kappa * kappa * t * t + 4.*M_PI * M_PI * ((double)n) * ((double)order_tr));

        tmp = tmp + pnl_rand_gamma(2.*kappa * theta / (sigma * sigma), 1., generator);
    }
}

```

```

//-----compute the rest

theta_p = 4.*theta * kappa / (sigma * sigma);
moy = theta_p * t * t * sigma * sigma / (4.*M_PI * M_PI * ((double) order_tr));
var = theta * pow(sigma * t / M_PI, 4.) / (24.* pow(((double) order_tr), 3.));

lambda_n = moy * moy / var;
gamma_n = var / moy;

tmp = tmp + pnl_rand_gamma(lambda_n, gamma_n, generator);

return tmp;
}

//-----Sample gthe law X1 by trancation series.
static double X_3_sample(double order_tr, double t, double kappa, double sigma,
{
//-----Declaration of variable
double lambda_n, moy, var;
double gamma_n;
int n;
double tmp;

//-----Compute the sum part

tmp = 0.;
for (n = 1; n <= order_tr; n++)
{

    gamma_n = (kappa * kappa * t * t + 4.*M_PI * M_PI * ((double)n) * ((double)

    tmp = tmp + pnl_rand_gamma(2., 1., generator) / gamma_n;
}

//-----compute the rest

```

```

moy = t * t * sigma * sigma / (M_PI * M_PI * ((double) order_tr));
var = pow(sigma * t / M_PI, 4.) / (6.* pow(((double) order_tr), 3.));

lambda_n = moy * moy / var;
gamma_n = var / moy;

tmp = tmp + pnl_rand_gamma(lambda_n, gamma_n, generator);

return tmp;
}

//-----Sampling the transition probability (v(0)=X_t, v(1)=in
//-----dX_t = kappa(theta-X_t)dt + sigma sqrt(X_t)dW_t
//-----In the case of the troncation serie
static void Sample_C(PnlVect *v, double t, double kappa, double sigma, double th
{
    //-----Declaration of variable
    double gamma, lambda;
    double tmp;
    double tmp2;
    int j, pss;
    int order_tr; // Default value is equal to 20
    //-----Initialization of parammter
    tmp = 0.;
    j = 0;
    gamma = 4.*kappa / (sigma * sigma * (1. - exp(-kappa * t)));
    lambda = pnl_vect_get(v, 0) * gamma * exp(-kappa * t);
    order_tr = 20;
    //-----Begin operations
    //----generate vt --> tmp
    pss = pnl_rand_poisson(lambda * 0.5, generator);

    for (j = 1; j <= pss; j++)
        tmp = tmp + pnl_rand_gamma(1., 2., generator);

    tmp = tmp + pnl_rand_gamma(2.*kappa * theta / (sigma * sigma), 2., generator);
    tmp = tmp / gamma;

```

```

//----generate the variable Z

tmp2 = 0.;
j = 0;
pss = pnl_rand_bessel(2.*theta * kappa / (sigma * sigma) - 1., 2.*kappa * sqrt

for (j = 1; j <= pss; j++)
{
    tmp2 = tmp2 + X_3_sample(order_tr, t, kappa, sigma, generator);
}
//----generate int_0^t vs = X1 +X2 +X3 --> lambda

lambda = tmp2 + X_2_sample(order_tr, t, kappa, sigma, theta , generator)

//----set the new value
pnl_vect_set(v, 0, tmp);
pnl_vect_set(v, 1, lambda);
}

int MCGlassermanKim(double S0, NumFunc_1 *p, double T, double r, double q, doub
{
    //-----Declaration of variable
    int j, call_put;
    PnlVect *vv;
    double tmp1, tmp, tmp2, tmp3;
    double tmpvar;
    int init_mc;
    double mt, sigmat;
    double d1, d2;
    double epsilon;
    double K;

    if ((p->Compute) == &Call)
        call_put = 0;
    else
        call_put = 1;
    K = p->Par[0].Val.V_PDOUBLE;

    //-----Initialization of variable
    vv = pnl_vect_create_from_double(2, 0.);
    pnl_vect_set(vv, 0, v0);

```

```

epsilon = 0.01;
init_mc = pnl_rand_init(generator, 1, (long)Nmc);
//-----Operation begins
tmp = 0.;
tmp2 = 0.;
tmpvar = 0.;
for (j = 1; j <= Nmc; j++)
{
    pnl_vect_set(vv, 0, v0);
    pnl_vect_set(vv, 1, 0.);

    Sample_C(vv, T, kappa, sigma, theta , generator);

    mt = (r - q - kappa * theta * rho / sigma) * T + rho * (pnl_vect_get(vv, 0

    sigmat = sqrt((1 - rho * rho) * pnl_vect_get(vv, 1));

    if (call_put == 0) //call pricing
    {
        d1 = (log(S0 / K) + mt) / sigmat + sigmat;
        d2 = d1 - sigmat;
        tmp1 = S0 * exp(mt - r * T + 0.5 * sigmat * sigmat) * cdf_nor(d1) - K

        d1 = (log((S0 + epsilon) / K) + mt) / sigmat + sigmat;
        d2 = d1 - sigmat;
        tmp3 = (S0 + epsilon) * exp(mt - r * T + 0.5 * sigmat * sigmat) * cdf_

    }
    else
    {
        d1 = (log(S0 / K) + mt) / sigmat + sigmat;
        d2 = d1 - sigmat;
        tmp1 = K * exp(-r * T) * (1. - cdf_nor(d2)) - S0 * (1. - exp(mt - r *

        d1 = (log((S0 + epsilon) / K) + mt) / sigmat + sigmat;
        d2 = d1 - sigmat;
        tmp3 = K * exp(-r * T) * (1. - cdf_nor(d2)) - (S0 + epsilon) * (1. -
    }
}

```



```

        ptMod->Rho.Val.V_PDOUBLE,
        Met->Par[0].Val.V_LONG, Met->Par[1].Val.V_ENUM.value,
        &(Met->Res[0].Val.V_DOUBLE),
        &(Met->Res[1].Val.V_DOUBLE),
        &(Met->Res[2].Val.V_DOUBLE)
    );
}

static int CHK_OPT(MC_GlassermanKim_Heston)(void *Opt, void *Mod)
{
    if ((strcmp(((Option *)Opt)->Name, "CallEuro") == 0) || (strcmp(((Option *)Opt)
        return OK;

    return WRONG;
}
#endif //PremiaCurrentVersion

static int MET(Init)(PricingMethod *Met, Option *Opt)
{
    //int type_generator;
    if (Met->init == 0)
    {
        Met->init = 1;

        Met->Par[0].Val.V_LONG = 50000;
        Met->Par[1].Val.V_ENUM.value = 0;
        Met->Par[1].Val.V_ENUM.members = &PremiaEnumMCRNGs;
    }
    return OK;
}

PricingMethod MET(MC_GlassermanKim_Heston) =
{
    "MC_GlassermanKim",
    { {"N iterations", LONG, {100}, ALLOW},
      {"RandomGenerator", ENUM, {100}, ALLOW},
      {" ", PREMIA_NULLTYPE, {0}, FORBID}
    },

```



```
CALC(MC_GlassermanKim_Heston),
{ {"Price", DOUBLE, {100}, FORBID},
  {"Delta", DOUBLE, {100}, FORBID} ,
  {"Error Price", DOUBLE, {100}, FORBID},
  {" ", PREMIA_NULLTYPE, {0}, FORBID}
},
CHK_OPT(MC_GlassermanKim_Heston),
CHK_mc,
MET(Init)
};
```