

Help

```

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

#if defined(PremiaCurrentVersion) && PremiaCurrentVersion < (2012+2) //The "#els
static int CHK_OPT(MC_Joshi)(void *Opt, void *Mod)
{
    return NONACTIVE;
}
int CALC(MC_Joshi)(void *Opt, void *Mod, PricingMethod *Met)
{
    return AVAILABLE_IN_FULL_PREMIA;
}
#else

//-----The calculus of Laplace Transform for the variable X1
static double pnl_Conf_Hyper_11(int orderTr, double a, double b, double z)
{
    int i;
    double id;
    double tmp1;
    double tmp2;

    tmp2 = a * z / b;
    tmp1 = tmp2 + 1.;
    for (i = 1; i <= orderTr; i++)
    {
        id = (double)i;
        tmp2 = tmp2 * (a + id) * z / ((b + id) * (id + 1.));
        tmp1 = tmp2 + tmp1;
    }
}

```

```

    return tmp1;
}

static int rand_bessel(double mu, double z, int generator)
{
    //-----Inittialization of variable
    double p0;
    double tmp, u;
    int n;
    //-----Begin operation
    p0 = pow(z * 0.5, mu) / (pnl_bessel_i(mu, z) * pnl_sf_gamma_inc(mu + 1., 0.));
    u = pnl_rand_uni(generator);
    tmp = 0;
    n = 0;
    if (u <= p0)
        return 0;
    do
    {
        tmp = tmp + p0;
        p0 = p0 * z * z / (4.*(((double)n) + 1.) * (((double)n) + 1. + mu));
        n++;
    }
    while ((u > tmp + p0));
    return n;
}

//-----Laplace transform  $E(\exp(-\int_0^T (d_1 X_s + d_2 X_s^{-1}) ds)$ 
//----- $dX_t = (a - bX_t)dt + c \sqrt{X_t}dW_t$ 
static double Lap_G(double x, double t, double a, double b, double c, double d1,
{
    //-----Declaration of parameter

    double alpha, beta;
    double v1, v2, gamma;
    int order = 50;

    //-----Initialization of parameter

    alpha = 2.*a / (c * c) - 1.;

    beta = 2.*b / (c * c);

```

```

v1 = 0.5 * (-beta + sqrt(beta * beta + 8.*d1 / (c * c)));

v2 = 0.5 * (-alpha + sqrt(alpha * alpha + 8.*d2 / (c * c)));

gamma = (beta + 2.*v1) * (1. - exp(-c * c * t * (v1 + 0.5 * beta)));

printf("fff \ n");

//-----Verification of the set of parameter
if (alpha + v2 + w2 + 1 <= 0 || gamma + w1 - v1 <= 0)
{
    printf("Error -- The set of parameter is not adequate for Laplace Transform");
    return 0;
}
else
{
    return exp(-(a * v1 + b * v2 + c * c * v1 * v2) * t) * pow(x, v2) * pow(gamma, w1) *
    pn1_Conf_Hyper_11(order, v2 - w2, alpha + 1. + 2.*v2, -gamma * gamma * x * exp(-gamma * t));
}

//-----Laplace transform E(exp(-int^T_0(d_1X_s+d_2X_s^-1)ds))
//-----dX_t = kappa(theta-X_t)dt + sigma sqrt(X_t)dW_t
double Lap_G1(double x, double t, double theta, double sigma, double kappa, double d1, double d2, double w1, double w2)
{
    return Lap_G(x, t, theta * kappa, kappa, sigma, d1, d2, w1, w2);
}

//-----Laplace transform E( exp(-lambda.X_T ))
//-----dX_t = kappa(theta-X_t)dt + sigma sqrt(X_t)dW_t
//-----In the case of lambda is complex value
/* fcomplex Lap_CIR(double x,double t, fcomplex lambda, double theta, double sigma)
/* { */
/* fcomplex lambda_p; */
/* double theta_p; */
/* double x_p ; */
/* fcomplex tmp1,tmp2; */
/* fcomplex tmp3; */
/* lambda_p = CRmul(lambda, sigma*sigma*(1.-exp(-kappa*t))/(4.*kappa)); */
/* theta_p = 4.* kappa*theta/(sigma*sigma); */
/* x_p = x*4.*kappa*exp(-kappa*t)/(sigma*sigma*(1.-exp(-kappa*t))); */

```

```

/*  tmp1 = RCmul(2.,lambda_p); */
/*  tmp2 = RCsub(1.,tmp1); */
/*  tmp1 = RCmul(x,lambda_p); */
/*  tmp3 = Cdiv(tmp1,tmp2); */
/*  tmp1 = Cexp(tmp3); */
/*  tmp3 = Cpow_real(tmp2,theta_p*0.5); */
/*  tmp2 = Cdiv(tmp1,tmp3); */
/*  return tmp2; */
/* } */
//-----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) * ((do

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

//-----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 = rand_bessel(2.*theta * kappa / (sigma * sigma) - 1., 2.*kappa * sqrt(pnl

    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 MCJoshi(double S0, NumFunc_1 *p, double T, double r, double q, double v0, d
{

//-----Declaration of variable
int j;
PnlVect *vv;
double tmp1, tmp, tmp2, tmp3;
double tmpvar;
int init_mc;
double mt, sigmat;
double epsilon;
int call_put;
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));

//d1 = (log(S0/K)+mt+sigmat*sigmat)/sigmat;

tmp1 = exp(mt + sigmat * pnl_rand_normal(generator));
tmp3 = (S0 + epsilon) * tmp1;
tmp1 = tmp1 * S0;

if (call_put == 0) //call pricing
{
    //tmp1= S0*exp(mt+0.5*sigma*sigma-r*T)*cdf_nor(d1)-K*exp(-r*T)*cdf_nor
    if (tmp1 >= K)
        tmp1 = tmp1 - K;
    else
        tmp1 = 0.;

    if (tmp3 >= K)
        tmp3 = tmp3 - K;
    else
        tmp3 = 0.;

}
else
{
    //tmp1 =( 1.-K*exp(-r*T))*cdf_nor(d2)-S0*(1.-cdf_nor(d1));
    if (tmp1 <= K)
        tmp1 = -tmp1 + K;
    else
        tmp1 = 0.;

    if (tmp3 <= K)
        tmp3 = -tmp3 + K;
    else
        tmp3 = 0.;
}

```

```

    tmp = tmp1 + tmp;
    tmp2 = tmp3 + tmp2;

    //-----confidence interval
    tmpvar = tmp1 * tmp1 + tmpvar;
}

tmp = exp(-r * T) * tmp / ((double)Nmc);
tmp2 = exp(-r * T) * tmp2 / ((double)Nmc);
tmpvar = tmpvar / ((double) Nmc) - tmp * tmp;

*ptprice = tmp;

*ptdelta = (tmp2 - tmp) / epsilon;

*pterror = sqrt(tmpvar / ((double) Nmc));

//-----Free Memory
pnl_vect_free(&vv);
return init_mc;
}

int CALC(MC_Joshi)(void *Opt, void *Mod, PricingMethod *Met)
{
    TYPEOPT *ptOpt = (TYPEOPT *)Opt;
    TYPEMOD *ptMod = (TYPEMOD *)Mod;
    double r, divid;

    r = log(1. + ptMod->R.Val.V_DOUBLE / 100.);
    divid = log(1. + ptMod->Divid.Val.V_DOUBLE / 100.);

    return MCJoshi(ptMod->S0.Val.V_PDOUBLE,
        ptOpt->PayOff.Val.V_NUMFUNC_1,
        ptOpt->Maturity.Val.V_DATE - ptMod->T.Val.V_DATE,
        r,
        divid, ptMod->Sigma0.Val.V_PDOUBLE
        , ptMod->MeanReversion.Val.V_PDOUBLE,
        ptMod->LongRunVariance.Val.V_PDOUBLE,
        ptMod->Sigma.Val.V_PDOUBLE,

```

```

        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_Joshi)(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;
        Met->HelpFilenameHint = "mc_joshi";

    }
    return OK;
}

PricingMethod MET(MC_Joshi) =
{
    "MC_Joshi",
    { {"N iterations", LONG, {100}, ALLOW},
      {"RandomGenerator", ENUM, {100}, ALLOW},

```

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