

Help

```
#include <stdlib.h>
#include "merhes1d_std.h"
#include "pnl/pnl_basis.h"
#include "pnl/pnl_vector.h"
#include "pnl/pnl_matrix.h"
#include "pnl/pnl_mathtools.h"
#include "pnl/pnl_specfun.h"

#if defined(PremiaCurrentVersion) && PremiaCurrentVersion < (2010+2) //The "#els
static int CHK_OPT(MC_Polynomial)(void *Opt, void *Mod)
{
    return NONACTIVE;
}
int CALC(MC_Polynomial)(void *Opt, void *Mod, PricingMethod *Met)
{
    return AVAILABLE_IN_FULL_PREMIA;
}
#else

/*Moments of normal distribution*/
static int moments_normal(PnlVect *c, int m, double mu, double gamma2)
{
    /* Input: PnlVect *c is of dimension m,
        mu=mean of normal distribution,
        gamma2=variance of normal distribution.
        The moments of the normal distribution with mean mu and variance gamma2 up
        degree m are calculated and stored in c.
    */
    int i, j, n, index1;
    PnlMat *a;

    index1 = (int)(m / 2 + 1.);
    LET(c, 0) = mu;

    a = pnl_mat_create_from_double(m, index1, 0.);

    for (j = 0; j < m; j++)
    {
```

```

        MLET(a, j, 0) = 1;
    }

    for (i = 2; i < m + 1; i++)
    {
        index1 = (int)(i / 2 + 1.);
        for (n = 2; n <= index1; n++)
        {
            MLET(a, i - 1, n - 1) = MGET(a, i - 2, n - 2) * (i - 1 - 2 * (n - 1) +
            LET(c, i - 1) = GET(c, i - 1) + MGET(a, i - 1, n - 1) * pow(mu, (double)
        }
        LET(c, i - 1) = GET(c, i - 1) + pow(mu, (double)i);
    }
    pnl_mat_free(&a);

    return 1.;
}

/*Matrix correponding to infinitesimal generator of the Bates model */
static int matrix_computation(PnlMat *A, int m, double r, double divid, double k
{
    /* Input: PnlMat *A is of dimension (m+1)(m+2)/2 x (m+1)(m+2)/2,
        m corresponds to the degree of the polynomial,
        other parameters are the inputs of the Bates model.
        The procedure calculates the matrix corresponding to the infinitesimal gener
        applied to the polynomials  $x^{iv^j}$ ,  $(i+j) \leq m$ , where x corresponds to the
        logprice and v to the variance. It is stored in A.
    */
    int i, j, k, row, column;
    PnlVect *c;

    c = pnl_vect_create_from_double(m, 0.);
    moments_normal(c, m, mu, gamma2);

    for (i = 0; i < m + 1; i++)
    {
        for (j = 0; j < m - i + 1; j++)
        {
            row = (i + j) * (i + j + 1) / 2 + j;
            for (k = 2; k < i + 1; k++)
            {

```

```

        column = (i - k + j) * (i - k + j + 1) / 2 + j;
        MLET(A, row, column) = (double) pnl_sf_choose(i, k) * GET(c, k - 1);
    }
    if (i > 0)
    {
        MLET(A, row, ((i - 1 + j) * (i + j) / 2 + j)) = i * (r - divid + 1);
        MLET(A, row, ((i + j) * (i + j + 1) / 2 + j + 1)) = -(double)i / 2;
    }
    MLET(A, row, ((i + j) * (i + j + 1) / 2 + j)) = -j * kappa;
    if (j > 0)
    {
        MLET(A, row, ((i + j - 1) * (i + j) / 2 + j - 1)) = j * (kappa * t);
    }
    if (i > 1)
    {
        MLET(A, row, ((i + j - 1) * (i + j) / 2 + j + 1)) = (double)i * ((i + j - 1) * (i + j) / 2 + j + 1);
    }
}

}

pnl_vect_free(&c);

return 1.;
}

/* Approximation of the call/put function by a polynomial*/
static int pol_approx(NumFunc_1 *p, PnlVect *coeff, double S0, double K, int Nb)
{
    /* Input: NumFunc_1 *p specifies the payoff function (Call or Put),
       PnlVect *coeff is of dimension Nb_Degree_Pol+1,
       S0 initial value to determine the interval where the approximation
       is done,
       K strike,
       Nb_Degree_Pol corresponds to the degree of the approximating
       polynomial.

       The coefficients of the polynomial approximating the Call/Put payoff
       function are calculated and stored in coeff in increasing order starting
       with the coefficient corresponding to degree 0.

    */
    PnlMat *x;
    PnlVect *y;
    PnlBasis *f;
    int dim;

```

```

int j;

dim = (int)((log(S0 * 10) - log(S0 / 10)) / 0.01 + 0.5); /* [log(S0/10), log(S0*10)]
the interval of approximation

x = pnl_mat_create_from_double(dim, 1, 0.);
y = pnl_vect_create_from_double(dim, 0.);

MLET(x, 0, 0) = log(S0 / 10);
LET(y, 0) = (p->Compute)(p->Par, S0 / 10.);

for (j = 1; j < dim; j++)
{
    MLET(x, j, 0) = MGET(x, j - 1, 0) + 0.01; /* grid of eq
with distance 0.01 in
[log(S0/10), log(S0*10)]

    LET(y, j) = (p->Compute)(p->Par, exp(MGET(x, j, 0))); /* evaluation of
function at x */
}

f = pnl_basis_create(PNL_BASIS_CANONICAL, Nb_Degree_Pol + 1, 1);
pnl_basis_fit_ls(f, coeff, x, y);

pnl_basis_free(&f);
pnl_mat_free(&x);
pnl_vect_free(&y);

return 1.;
}

/* European Call/Put price with Bates model */
int MCCuchieroKellerResselTeichmann(double S0, NumFunc_1 *p, double t, double r)
{
    /* Inputs: NumFunc_1 *p specifies the payoff function (Call or Put),
    S_0, K, t: option inputs
    other inputs: model parameters
    N: number of iterations in the Monte Carlo simulation
    M: number of discretization steps
    Nb_Degree_Pol: degree of approximating polynomial for variance
    reduction, between 0 and 8.
    Calculates the price of a European Call/Put option using Monte Carlo

```

with variance reduction. Variance reduction is achieved by approximating the payoff function with a polynomial whose expectation can be calculated analytically and which therefore serves as control variate.

*/

```
double mean_price, var_price, mean_delta, mean_delta_novar, var_delta, polypri;
double poly_sample, price_sample_novar, delta_sample, delta_poly_sample, delta;
int init_mc, i, j, k, n, m1, m2;
int simulation_dim = 1;
double alpha, z_alpha;
double g, g2, h, Xt, Vt;
double dt, sqrt_dt;
int nj;
double poisson_jump = 0, mm = 0, Eu;
int matrix_dim, line;
double K;
double rhoc;
PnlMat *A, *eA;
PnlVect *coeff;
PnlVect *deltacoeff;
PnlVect *matcoeff;
PnlVect *deltamatcoeff;
PnlVect *b;
PnlVect *deltab;
PnlVect *veczero;
PnlVect *vecX;

rhoc = sqrt(1. - SQR(rho));
Eu = exp(mu + 0.5 * gamma2) - 1.;
dt = t / (double)M;
sqrt_dt = sqrt(dt);

K = p->Par[0].Val.V_DOUBLE;

/* Initialisation of vectors and matrices */
coeff = pnl_vect_create_from_double(Nb_Degree_Pol + 1, 0.);
deltacoeff = pnl_vect_create_from_double(Nb_Degree_Pol + 1, 0.);
vecX = pnl_vect_create_from_double(Nb_Degree_Pol + 1, 0.);

matrix_dim = (Nb_Degree_Pol + 1) * (Nb_Degree_Pol + 2) / 2;
matcoeff = pnl_vect_create_from_double(matrix_dim, 0.);
```

```

deltamatcoeff = pnl_vect_create_from_double(matrix_dim, 0.);
veczero = pnl_vect_create_from_double(matrix_dim, 0.);
A = pnl_mat_create_from_double(matrix_dim, matrix_dim, 0.);
eA = pnl_mat_create_from_double(matrix_dim, matrix_dim, 0.);

/* Approximation of payoff function */
pol_approx(p, coeff, S0, K, Nb_Degree_Pol);

/* Calculation of the coefficients of the derivative of the approximating
   polynomial */
for (n = 0; n < Nb_Degree_Pol; n++)
    LET(deltacoeff, n) = GET(coeff, n + 1) * (double)n;

/* Reordering of coefficients, to fit the size of the generator matrix */
for (n = 0; n < Nb_Degree_Pol + 1; n++)
{
    LET(matcoeff, n * (n + 1) / 2) = GET(coeff, n);
    LET(deltamatcoeff, n * (n + 1) / 2) = GET(deltacoeff, n);
}

/* Calculation of the matrix corresponding to the generator of the Bates model
matrix_computation(A, Nb_Degree_Pol, r, divid, kappa, theta, lambda, mu, gamma
pnl_mat_mult_double(A, t);

/* Matrix exponentiation */
pnl_mat_exp(eA, A);

pnl_mat_sq_transpose(eA);
b = pnl_mat_mult_vect(eA, matcoeff);
deltab = pnl_mat_mult_vect(eA, deltamcoeff);

/* Calculation  $\log(S0)^{m1} V0^{m2}$ ,  $m1+m2 \leq Nb\_Degree\_Pol$  */
for (m1 = 0; m1 < Nb_Degree_Pol + 1; m1++)
    for (m2 = 0; m2 < Nb_Degree_Pol + 1 - m1; m2++)
    {
        line = (m1 + m2) * (m1 + m2 + 1) / 2 + m2;
        LET(veczero, line) = pow(log(S0), (double) m1) * pow(V0, (double) m2);
    }

/* Expectation of approximating polynomial */
polyprice = pnl_vect_scalar_prod(b, veczero);

```

```
/* Expectation of derivative of approximating polynomial */
polydelta = pnl_vect_scalar_prod(deltab, veczero);

/* Value to construct the confidence interval */
alpha = (1. - confidence) / 2.;
z_alpha = pnl_inv_cdfnor(1. - alpha);

/*Initialisation*/
mean_price = 0.0;
var_price = 0.0;
mean_delta = 0.0;
mean_delta_novar = 0.0;
var_delta = 0.0;
mean_price_novar = 0.;

/*MC sampling*/
init_mc = pnl_rand_init(generator, simulation_dim, N);

/* Test after initialization for the generator */
if (init_mc == OK)
{
    /* Begin N iterations */
    for (i = 0; i < N; i++)
    {
        Xt = log(S0);
        Vt = V0;
        for (j = 0; j < M; j++)
        {
            mm = r - divid - 0.5 * Vt - lambda * Eu;

            /* Generation of standard normal random variables */
            g = pnl_rand_normal(generator);
            g2 = pnl_rand_normal(generator);

            /* Generation of Poisson random variable */
            if (pnl_rand_uni(generator) < lambda * dt)
                nj = 1;
            else nj = 0;
        }
    }
}
```

```

/* Normally distributed jump size */
h = pnl_rand_normal(generator);
poisson_jump = mu * nj + sqrt(gamma2 * nj) * h;

/* Euler scheme for log price and variance */
Xt += mm * dt + sqrt(MAX(0., Vt)) * sqrt_dt * g + poisson_jump;
Vt += kappa * (theta - Vt) * dt + sigmav * sqrt(MAX(0., Vt)) * sqrt_dt *
      (rho * g + rhoc * g2);
}
price_sample_novar = (p->Compute)(p->Par, exp(Xt));

/* Creation of vector containing  $X_t^k$ ,  $k \leq \text{Nb\_Degree\_Pol}$  */
for (k = 0; k < Nb_Degree_Pol + 1; k++)
{
    LET(vecX, k) = pow(Xt, (double) k);
}

/* Approximating polynomial evaluated at  $X_t$  */
poly_sample = pnl_vect_scalar_prod(coeff, vecX);
/* Derivative of approximating polynomial evaluated at  $X_t$  */
delta_poly_sample = pnl_vect_scalar_prod(deltacoeff, vecX);

/*Sum for prices*/
mean_price_novar += price_sample_novar; /* without control variate */
mean_price += price_sample_novar - (poly_sample - polyprice); /* with

/* Delta sampling */
if (price_sample_novar > 0.)
{
    delta_sample_novar = exp(Xt) / S0; //Call Case
    delta_sample = (exp(Xt) - (delta_poly_sample - polydelta)) / S0;
    if ((p->Compute) == &Put)
        delta_sample = (-exp(Xt) - (delta_poly_sample - polydelta)) / S0
}

else
{
    delta_sample_novar = 0;
    delta_sample = 0. - (delta_poly_sample - polydelta) / S0; /* Delta

```



```

        /*Sum for delta*/
        mean_delta_novar += delta_sample_novar;    /* without control variate*/
        mean_delta += delta_sample;                /* with control variate*/

        /*Sum of squares*/
        var_price += SQR(price_sample_novar - (poly_sample - polyprice));
        var_delta += SQR(delta_sample);
    }

    /*Price */
    *ptprice = exp(-r * t) * (mean_price / (double) N);

    /*Error*/
    *pterror_price = sqrt(exp(-2.0 * r * t) * var_price / (double)N - SQR(*ptprice));

    /* Price Confidence Interval */
    *inf_price = *ptprice - z_alpha * (*pterror_price);
    *sup_price = *ptprice + z_alpha * (*pterror_price);

    /*Delta*/
    *ptdelta = exp(-r * t) * mean_delta / (double) N;
    *pterror_delta = sqrt(exp(-2.0 * r * t) * (var_delta / (double)N - SQR(*ptdelta)));

    /* Delta Confidence Interval */
    *inf_delta = *ptdelta - z_alpha * (*pterror_delta);
    *sup_delta = *ptdelta + z_alpha * (*pterror_delta);
}

//Memory desallocation
pnl_mat_free(&eA);
pnl_mat_free(&A);
pnl_vect_free(&coeff);
pnl_vect_free(&b);
pnl_vect_free(&matcoeff);
pnl_vect_free(&veczero);
pnl_vect_free(&vecX);
pnl_vect_free(&deltacoeff);
pnl_vect_free(&deltamatcoeff);
pnl_vect_free(&deltab);

```

```

    return init_mc;
}

int CALC(MC_Polynomial)(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 MCCuchieroKellerResselTeichmann(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,
                                             ptMod->Mean.Val.V_PDOUBLE,
                                             ptMod->Variance.Val.V_PDOUBLE,
                                             ptMod->Lambda.Val.V_PDOUBLE,
                                             Met->Par[0].Val.V_LONG,
                                             Met->Par[1].Val.V_INT,
                                             Met->Par[2].Val.V_INT,
                                             Met->Par[3].Val.V_ENUM.value,
                                             Met->Par[4].Val.V_PDOUBLE,
                                             &(Met->Res[0].Val.V_DOUBLE),
                                             &(Met->Res[1].Val.V_DOUBLE),
                                             &(Met->Res[2].Val.V_DOUBLE),
                                             &(Met->Res[3].Val.V_DOUBLE),
                                             &(Met->Res[4].Val.V_DOUBLE),
                                             &(Met->Res[5].Val.V_DOUBLE),
                                             &(Met->Res[6].Val.V_DOUBLE),
                                             &(Met->Res[7].Val.V_DOUBLE));
}

```

```

static int CHK_OPT(MC_Polynomial)(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 = 15000;
        Met->Par[1].Val.V_INT = 100;
        Met->Par[2].Val.V_INT = 4;
        Met->Par[3].Val.V_ENUM.value = 0;
        Met->Par[3].Val.V_ENUM.members = &PremiaEnumMCRNGs;
        Met->Par[4].Val.V_DOUBLE = 0.95;
    }

    return OK;
}

PricingMethod MET(MC_Polynomial) =
{
    "MC_Polynomial",
    { {"N iterations", LONG, {100}, ALLOW},
      {"M TimeStepNumber", LONG, {100}, ALLOW},
      {"Polynomial Degree", INT, {100}, ALLOW},
      {"RandomGenerator", ENUM, {100}, ALLOW},
      {"Confidence Value", DOUBLE, {100}, ALLOW},
      {" ", PREMIA_NULLTYPE, {0}, FORBID}

```

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