Help

```
#include "
href../../../mod/fps1d/fps1d_std/fps1d_std_h_src.pdffps1d_std.h"
#include "
href../../../common/enums_h_src.pdfenums.h"
#include "pnl/pnl_cdf.h"

#if defined(PremiaCurrentVersion) && PremiaCurrentVersion < (2007+2) //The "#els
static int CHK_OPT(MC_FPS)(void *Opt, void *Mod)
{
  return NONACTIVE;
}
int CALC(MC_FPS)(void *Opt, void *Mod, PricingMethod *Met)
{
  return AVAILABLE_IN_FULL_PREMIA;
}
#else

/*
 * in-place Choleski decomposition
 */
static void cholesky(int n, double **s)
{
  int i, j, k;

  for (k = 0; k < n; k++)
    {
      s[k][k] = sqrt(s[k][k]);
      for (i = k + 1; i < n; i++)
        {
          s[i][k] *= (1.0 / s[k][k]);
          for (j = k + 1; j <= i; j++)
            s[i][j] -= (s[i][k] * s[j][k]);
        }
    }
  for (i = 0; i < n; i++)
    for (j = i + 1; j < n; j++)
      s[i][j] = 0.0;
}
```

```
/*Covariance Matrix*/
static double **covar(double MM, double T, double alpha, double nu)
{

  double h = T * (1.0 / MM);
  double beta = nu * sqrt(2.0 * alpha);
  double **C;

  C = malloc(2 * sizeof(double *));
  C[0] = malloc(2 * sizeof(double));
  C[1] = malloc(2 * sizeof(double));

  C[0][0] = nu * nu * (exp(2 * alpha * h) - 1);
  C[0][1] = (beta * (1.0 / alpha)) * (exp(alpha * h) - 1);
  C[1][0] = (beta * (1.0 / alpha)) * (exp(alpha * h) - 1);
  C[1][1] = h;

  return C;
}

/* Function useful for Simulation of pair (Y,W_2)
 * be sure that res is already mallocated
 */
static void g(double *res, double v, double w, double u1, double u2, double MM,
{
  double h = T * (1.0 / MM);

  res[0] = (v + u1) * exp(- alpha * h);
  res[1] = w + u2;

}

/*Simulation of pair (Y,W_2)*/
static double **V(int generator, double MM, double T,  double alpha, double nu,
{
  int i;
  double R = MM + 1;
  double h = T * (1.0 / MM);
  double **v;
  double **U;
```

```
double **C;
double *u;
double *q;
double *p;

v = malloc(R * sizeof(double *));
U = malloc(R * sizeof(double *));
u = malloc(2 * sizeof(double));
p = malloc(2 * sizeof(double));
q = malloc(2 * sizeof(double));
C = covar(MM, T, alpha, nu);
cholesky(2, C);
v[0] = malloc(2 * sizeof(double));
v[0][0] = Y0;
v[0][1] = 0.0;
p[0] = pnl_rand_normal(generator);
p[1] = pnl_rand_normal(generator);

for (i = 1; i <= MM; i++)
  {
    v[i] = malloc(2 * sizeof(double));
    U[i] = malloc(2 * sizeof(double));

    if ((i % 2) == 0)
      v[i - 1][1] += p[0] * sqrt(h);
    if ((i % 2) != 0)
      {
        v[i - 1][1] += p[1] * sqrt(h);
        p[0] = pnl_rand_normal(generator);
        p[1] = pnl_rand_normal(generator);
      }


    v[i][1] = v[i - 1][1];
    q[0] = pnl_rand_normal(generator);
    q[1] = pnl_rand_normal(generator);

    U[i][0] = ((C[0][0]) * (q[0])) + ((C[0][1]) * (q[1]));
    U[i][1] = ((C[1][0]) * (q[0])) + ((C[1][1]) * (q[1]));
    g(v[i], v[i - 1][0], v[i - 1][1] , U[i][0] , U[i][1] , MM, T, alpha);
  }
```

```
  free(u);
  free(p);
  free(q);

  free(C[0]);
  free(C[1]);
  free(C);
  for (i = 1; i <= MM; i++)
    free(U[i]);
  free(U);

  return v;
}

/* Stochastic volatility Function*/
static double f(double Y, double sigmaf)
{
  return exp(Y) * sigmaf;

}

/*
 * Simulation of stock process until maturity
 */
static void XT(double *xf, double MM, double *B, double **c, double T, double al
{
  double x;
  double xx;
  int i;
  double h = T * (1.0 / MM);
  double I0 = 0.0;
  double I1 = 0.0;
  double I1bis = 0.0;
  double I2 = 0.0;
  double I0bis = 0.0;
  double I2bis = 0.0;

  if (xf == NULL)
    xf = malloc(2 * sizeof(double));
  for (i = 0; i < (2 * (MM)); i++)
    {
```

```
      I0bis += (f(c[i][0], sigmaf) * f(c[i][0], sigmaf));
      I1bis += (f(c[i][0], sigmaf) * (B[i + 1]));
      I2bis += (f(c[i][0], sigmaf) * (c[i + 1][1] - c[i][1]));
      if ((i % 2) == 0)
        {
          I0 += (f(c[i][0], sigmaf) * f(c[i][0], sigmaf));
          I1 += (f(c[i][0], sigmaf) * (B[i + 2] + B[i + 1]));
          I2 += (f(c[i][0], sigmaf) * (c[i + 2][1] - c[i][1]));
        }

    }
  I0bis = I0bis * h * 0.5;
  I0    = I0 * h;
  x     = X * exp((r - divid) * T) * exp((rho * I2) + (sqrt(1.0 - (rho * rho)) *
  xx    = X * exp((r - divid) * T) * exp((rho * I2bis) + (sqrt(1.0 - (rho * rho)
  xf[0] = x;
  xf[1] = xx;

}


/* Computation of Price and Delta with MM and 2*MM steps of discretization of ED
   to obtain Romberg extrapolation*/
static int prix_es_delta(int generator, NumFunc_1  *p, double MM, double T, doub
{
  double h = T * (1.0 / MM);
  double *xf, *xf1;
  double *xxf, *xxf1;
  /* double *g;*/
  double x1 = 0.0;
  double xr1 = 0.0;
  double xx1 = 0.0;
  double xxr1 = 0.0;
  double x2 = 0.0;
  double payoff;
  double payoffr;
  double ppayoff;
  double ppayoffr;
  int i;
  int k;
```

```
double *B;
double *BB;
double *q;
double **c = NULL;
double x2delta = 0.0, delta, deltar;

/*Memory allocation*/
q = malloc(2 * sizeof(double));
xf = malloc(2 * sizeof(double));
xf1 = malloc(2 * sizeof(double));
xxf = malloc(2 * sizeof(double));
xxf1 = malloc(2 * sizeof(double));
B = malloc(((2 * MM) + 1) * sizeof(double));
BB = malloc(((2 * MM) + 1) * sizeof(double));

/*Simulation of S and Y*/
/*Antithetic Control Variate*/
q[0] = pnl_rand_normal(generator);
q[1] = pnl_rand_normal(generator);
for (i = 0; i < N; i++)
  {
    c = V(generator, 2 * (MM), T,  alpha,  nu, Y0);
    for (k = 0; k <= 2 * (MM); k++)
      {
        if ((k % 2) == 0)
          B[k] = q[0] * sqrt(h * 0.5);
        if ((k % 2) != 0)
          {
            B[k] = q[1] * sqrt(h * 0.5);
            q[0] = pnl_rand_normal(generator);
            q[1] = pnl_rand_normal(generator);
          }
        BB[k] = (-1.0) * B[k];
      }

    XT(xf, MM, B, c, T, alpha, nu, rho, r, divid, X, sigmaf);
    XT(xf1, MM, BB, c, T,  alpha,  nu, rho, r, divid, X, sigmaf);
    XT(xxf, MM, B, c, T, alpha, nu, rho, r, divid, X * (1 + pas), sigmaf);
    XT(xxf1, MM, BB, c, T,  alpha,  nu, rho, r, divid, X * (1 + pas), sigmaf);

    payoff  = 0.5 * ((p->Compute)(p->Par, xf[0]) + (p->Compute)(p->Par, xf1[0]
```

```
    payoffr  = 0.5 * ((p->Compute)(p->Par, xf[1]) + (p->Compute)(p->Par, xf1[1
    ppayoff  =  0.5 * ((p->Compute)(p->Par, xxf[0]) + (p->Compute)(p->Par, xxf
    ppayoffr =  0.5 * ((p->Compute)(p->Par, xxf[1]) + (p->Compute)(p->Par, xxf

    /*Price and Price Inc*/
    x1  += payoff;
    xr1 += payoffr;

    xx1  += ppayoff;
    xxr1 += ppayoffr;


    /*Delta*/
    delta = (ppayoff - payoff) / (X * pas);
    deltar = (ppayoffr - payoffr) / (X * pas);

    /*Sum of squares*/
    x2  += ((2.0 * payoffr) - payoff) * ((2.0 * payoffr) - payoff);
    x2delta  += ((2.0 * deltar) - delta) * ((2.0 * deltar) - delta);

    /* free v */
    for (k = 0; k <= 2 * MM; k++)
      free(c[k]);
    free(c);
  }
/*price with M steps*/
x1  *= (exp(-r * T) * (1.0 / N));

/*price with 2*M steps */
xr1 *= (exp(-r * T) * (1.0 / N));

/*Romberg extrapolation*/
xr1  = (2.0 * xr1) - x1;

/*price inc with M steps*/
xx1  *= (exp(-r * T) * (1.0 / N));
xxr1 *= (exp(-r * T) * (1.0 / N));

/*Romberg extrapolation inc */
xxr1  = (2.0 * xxr1) - xx1;
```

```
  /*delta*/
  xxr1 = (xxr1 - xr1) / (X * pas);

  /*error price*/
  x2  *= (exp(-2.0 * r * T) * (1.0 / N));
  x2   = x2 - (xr1 * xr1);
  x2 = sqrt(x2);
  x2 =  2.0 * x2 * (1.0 / sqrt(N));

  /*error delta*/

  x2delta  *= (exp(-2.0 * r * T) * (1.0 / N));
  x2delta   = x2delta - (xxr1 * xxr1);
  x2delta = sqrt(x2delta);
  x2delta =  2.0 * x2delta * (1.0 / sqrt(N));


  /*Values*/
  *p1 = xr1;
  *delta1 = xxr1;
  *error_price1 = x2;
  *error_delta1 = x2delta;

  /*Memory Desallocation*/
  free(xf);
  free(xxf);
  free(xf1);
  free(xxf1);
  free(q);
  free(B);
  free(BB);

  return OK;
}

static int MCFPS(double s, NumFunc_1  *p, double t, double r, double divid, doub
{
  int init_mc;
  int simulation_dim = 1, MM;
  double alpha1, z_alpha, inc = 0.0001;
  double es = 0.001;
```

```
double p1, p2, delta1, error_price1, delta2, error_price2, error_delta2;
double erreurt;

/* Value to construct the confidence interval */
alpha1 = (1. - confidence) / 2.;
z_alpha = pnl_inv_cdfnor(1. - alpha1);

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

/* Test after initialization for the generator */
if (init_mc == OK)
  {

    /*First Step*/
    MM = 1;
    prix_es_delta(generator, p, MM, t, alpha, nu, rho, r, divid, nb, y0, s, si
    prix_es_delta(generator, p, 2 * MM, t, alpha, nu, rho, r, divid, nb, y0, s
                  &p2, &delta2, &error_price2, &error_delta2);
    erreurt = (4.0 / 3.0) * (p1 - p2);

    /*Iterative Algorithm*/
    while ((fabs(erreurt) > 1.5 * fabs(es)) && (MM < 8))
      {
        MM  = 2 * MM;
        p1 = p2;
        prix_es_delta(generator, p, 2 * MM, t, alpha, nu, rho, r, divid, nb, y
                      &p2, &delta2, &error_price2, &error_delta2);
        erreurt = (4.0 / 3.0) * (p1 - p2);
      }

    /* Price estimator */
    *ptprice = p2;
    *pterror_price = error_price2;

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

    /* Delta estimator */
    *ptdelta = delta2;
```

```
      *pterror_delta = error_delta2;

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


int CALC(MC_FPS)(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 MCFPS(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->Rho.Val.V_PDOUBLE,
               ptMod->SigmaF.Val.V_PDOUBLE,
               Met->Par[0].Val.V_LONG,
               Met->Par[1].Val.V_ENUM.value,
               Met->Par[2].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_FPS)(void *Opt, void *Mod)
{
  Option *ptOpt = (Option *)Opt;
  TYPEOPT *opt = (TYPEOPT *)(ptOpt->TypeOpt);

  if ((opt->EuOrAm).Val.V_BOOL == EURO)
    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_ENUM.value = 0;
      Met->Par[1].Val.V_ENUM.members = &PremiaEnumMCRNGs;
      Met->Par[2].Val.V_DOUBLE = 0.95;


    }
  type_generator = Met->Par[1].Val.V_ENUM.value;

  if (pnl_rand_or_quasi(type_generator) == PNL_QMC)
    {
      Met->Res[2].Viter = IRRELEVANT;
      Met->Res[3].Viter = IRRELEVANT;
      Met->Res[4].Viter = IRRELEVANT;
      Met->Res[5].Viter = IRRELEVANT;
      Met->Res[6].Viter = IRRELEVANT;
      Met->Res[7].Viter = IRRELEVANT;
```

```
    }
  else
    {
      Met->Res[2].Viter = ALLOW;
      Met->Res[3].Viter = ALLOW;
      Met->Res[4].Viter = ALLOW;
      Met->Res[5].Viter = ALLOW;
      Met->Res[6].Viter = ALLOW;
      Met->Res[7].Viter = ALLOW;
    }
  return OK;
}


PricingMethod MET(MC_FPS) =
{
  "MC_FPS",
  { {"N iterations", LONG, {100}, ALLOW},
    {"RandomGenerator", ENUM, {100}, ALLOW},
    {"Confidence Value", DOUBLE, {100}, ALLOW},
    {" ", PREMIA_NULLTYPE, {0}, FORBID}
  },
  CALC(MC_FPS),
  { {"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_FPS),
  CHK_mc,
  MET(Init)
};
```