Help

```
#include  "hescir1d_std.h"
#include "enums.h"
#include "error_msg.h"

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



void static funcD1(dcomplex u, double lambda, double eta, dcomplex *result)
{
  if (lambda == 0.0 || eta == 0.0) *result = CZERO;
  else *result = Csqrt(RCadd(lambda * lambda, RCmul(2.*eta * eta, RCsub(1., Cmul
}
void static funcD2(dcomplex u, double gamma1, double rho12, double k, dcomplex *
{
  if (k == 0.0 || gamma1 == 0.0) *result = CZERO;
  else *result = Csqrt(Csub(Cpow_real(CRsub(RCmul(gamma1 * rho12, Cmul(CI, u)),
}
void static funcG1(double lambda, dcomplex D1, dcomplex *result)
{
  if (Creal(D1) == 0.0 && Cimag(D1) == 0.0) *result = CZERO;
  else *result = Cdiv(RCsub(lambda, D1), RCadd(lambda, D1));
}
void static funcG2(dcomplex u, double k, double gamma1, double rho12, dcomplex D
{
  if (Cimag(D2) == 0.0 && Creal(D2) == 0.0) *result = CZERO;
  else *result = Cdiv(Csub(RCsub(k, RCmul(gamma1 * rho12, Cmul(CI, u))), D2), Ca
}
void static funcBx(dcomplex u, dcomplex *result)
```

```
{
  *result = Cmul(u, CI);
}
void static funcBr(dcomplex u, double tau, double lambda, double eta, dcomplex D
{
  if (eta == 0.0)
    {
      if (lambda == 0.0) *result = RCmul(tau, CRsub(Cmul(u, CI), 1.0));
      else *result = RCmul((1.0 - exp(-lambda * tau)) / lambda, CRsub(Cmul(u, CI
    }
  else *result = Cmul(Cdiv(RCsub(1., Cexp(RCmul(-tau, D1))), RCmul(eta * eta, RC
}
void static funcBsigma(dcomplex u, double tau, double k, double gamma1, double r
{
  if (gamma1 == 0.0)
    {
      if (k == 0.0) *result = RCmul(-tau / 2.0, Cadd(Cmul(u, u), Cmul(u, CI)));
      else *result = RCmul((exp(-k * tau) - 1.0) / (2.0 * k), Cadd(Cmul(u, u), C
    }
  else *result = Cmul(Cdiv(RCsub(1., Cexp(RCmul(-tau, D2))) , RCmul(gamma1 * gam
}
void static funcC(double t, double speed, double variance, double *result)
{
  if (speed == 0.0 || variance == 0.0) *result = 0.0;
  else *result = variance * variance * (1. - exp(-speed * t)) / (4.*speed);
}
void static funcLambda(double t, double speed, double initial, double variance,
{
  if (speed == 0.0 || variance == 0.0) *result = 0.0;
  else *result = 4.*speed * initial * exp(-speed * t) / (variance * variance * (
}
void static funcD(double speed, double mean, double variance, double *result)
{
  if (speed == 0.0 || variance == 0.0) *result = 0.0;
  else *result = 4.*speed * mean / (variance * variance);
}
void static Lambda1(double t, double C, double Lambda, double D, double *result)
{
  if (D == 0.0 || Lambda == 0.0) *result = 0.0;
  else *result = sqrt(C * (Lambda - 1.) + C * D + C * D / (2.*(D + Lambda)));
}
```

```
void static contA(double mean, double speed, double variance, double *result)
{
  if (speed == 0.0 || variance == 0.0) *result = 0.0;
  else *result = sqrt(mean - variance * variance / 8. / speed);
}
void static contB(double initial, double A, double *result)
{
  *result = sqrt(initial) - A;
}
void static contC(double A, double B, double Lambda1, double *result)
{
  if (B == 0.0 || (Lambda1 - A) == 0.0) *result = 0.0;
  else *result = -log((Lambda1 - A) / B);
}
void static Expect(double A, double B, double C, double t, double *result)
{
  *result = A + B * exp(-C * t);
}
void static funcA(dcomplex u, double tau, double k, double sigmamean, double gam
{
  double expect1 = 0.0, expect2 = 0.0;
  dcomplex A = CZERO;
  Expect(a1, b1, c1, 0, &expect1);
  Expect(a2, b2, c2, 0, &expect2);
  if (lambda == 0.0 || eta == 0.0)
    {
      A = RCmul(k * sigmamean / pow(gamma1, 2.), Csub(RCmul(tau, RCsub(k, Cadd(R
    }
  else
    {
      A = RCmul(k * sigmamean / pow(gamma1, 2.), Csub(RCmul(tau, RCsub(k, Cadd(R
      A = Cadd(A, RCmul(lambda * theta / pow(eta, 2.), Csub(RCmul(tau, RCsub(lam
      A = Cadd(A, Cmul(RCmul(rho13 * eta * (expect1) * (expect2), Cmul(CI, u)),
    }
  *result = A;
}
void static func_dc(double t, double variance, double speed, double *result)
{
  if (speed == 0.0 || variance == 0.0) *result = 0.0;
  else *result = pow(variance, 2.) * exp(-speed * t) / 4.;
}
```

```
void static func_delambda(double t, double initial, double speed, double varianc
{
  if (speed == 0.0 || variance == 0.0) *result = 0.0;
  else *result = -4.* initial * pow(speed, 2.) * exp(speed * t) / (pow(exp(speed
}
void static func_miu(double t, double initial, double c, double d, double lambda
{
  if (t == 0.0) *result = sqrt(initial);
  else if (d + lambda == 0.0 || c == 0.0)
    {
      *result = 0.0;
    }
  else
    {
      *result = (c * (2. - d * pow((d + lambda), -2.)) * dlambda + (-2. + 2.*lam
    }
}
void static func_psi(double t, double c, double d, double lambda, double dc, dou
{
  if (t == 0.0) *result = 0.0;
  else if (d + lambda == 0.0)
    {
      *result = 0.0;
    }
  else
    {
      *result = 1. / sqrt(2.) * sqrt(fabs(((d + lambda) * (d + 2.*lambda) * dc +
    }
}

//Stochastic approximation
void static ode_bound(dcomplex u, PnlVectComplex *ybound, PnlVectComplex *dyboun
{
  pnl_vect_complex_set(ybound, 0, Cmul(u, CI));
  pnl_vect_complex_set(dybound, 1, RCadd(-1., Cmul(u, CI)));
  pnl_vect_complex_set(dybound, 5, Cmul(RCmul(0.5, Cmul(u, CI)), CRsub(Cmul(u, C
}
void static derivs(double tau, double maturity, double lambda, double theta, dou
{
  double c_v, c_r, d_v, d_r, lambda_v, lambda_r, dc_v, dc_r, dlambda_v, dlambda_
  funcC(maturity - tau, k, gamma1, &c_v);
```

```
  funcC(maturity - tau, lambda, eta, &c_r);
  funcLambda(maturity - tau, k, sigma0, gamma1, &lambda_v);
  funcLambda(maturity - tau, lambda, r0, eta, &lambda_r);
  funcD(k, sigmamean, gamma1, &d_v);
  funcD(lambda, theta, eta, &d_r);
  func_dc(maturity - tau, gamma1, k, &dc_v);
  func_dc(maturity - tau, eta, lambda, &dc_r);
  func_delambda(maturity - tau, sigma0, k, gamma1, &dlambda_v);
  func_delambda(maturity - tau, r0, lambda, eta, &dlambda_r);
  func_miu(maturity - tau, sigma0, c_v, d_v, lambda_v, dc_v, dlambda_v, &miu_v);
  func_miu(maturity - tau, r0, c_r, d_r, lambda_r, dc_r, dlambda_r, &miu_r);
  func_psi(maturity - tau, c_v, d_v, lambda_v, dc_v, dlambda_v, &psi_v);
  func_psi(maturity - tau, c_r, d_r, lambda_r, dc_r, dlambda_r, &psi_r);
  pnl_vect_complex_set(dy, 1, Cadd(Cadd(Cadd(RCadd(-1., pnl_vect_complex_get(y,
  pnl_vect_complex_set(dy, 2, Cadd(Cadd(RCmul(miu_v, pnl_vect_complex_get(y, 3))
  pnl_vect_complex_set(dy, 3, Cadd(Cadd(RCmul(eta * rho13, Cmul(pnl_vect_complex
  pnl_vect_complex_set(dy, 4, Cadd(Cadd(Csub(RCmul(1. / 2., Cmul(pnl_vect_comple
  pnl_vect_complex_set(dy, 5, Cadd(Cadd(Cadd(RCmul(miu_r, pnl_vect_complex_get(y
  pnl_vect_complex_set(dy, 6, Cadd(Cadd(Cadd(RCmul(k * sigmamean, pnl_vect_compl
}
void static ode_solution_step(dcomplex u, double maturity, double lambda, double
{

  int i;
  double xh, hh, h6;
  PnlVectComplex *dym, *dyt, *yt;
  dym = pnl_vect_complex_create_from_dcomplex(n, Complex(0.0, 0.0));
  dyt = pnl_vect_complex_create_from_dcomplex(n, Complex(0.0, 0.0));
  yt = pnl_vect_complex_create_from_dcomplex(n, Complex(0.0, 0.0));
  hh = step * 0.5;
  h6 = step / 6.0;
  xh = bound + hh;
  for (i = 0; i < n; i++) pnl_vect_complex_set(yt, i, Cadd(pnl_vect_complex_get(
  derivs(xh, maturity, lambda, theta, eta, k, sigmamean, gamma1, r0, sigma0, rho
  for (i = 0; i < n; i++) pnl_vect_complex_set(yt, i, Cadd(pnl_vect_complex_get(
  derivs(xh, maturity, lambda, theta, eta, k, sigmamean, gamma1, r0, sigma0, rho
  for (i = 0; i < n; i++)
    {
      pnl_vect_complex_set(yt, i, Cadd(pnl_vect_complex_get(ybound, i), RCmul(st
      pnl_vect_complex_set(dym, i, Cadd(pnl_vect_complex_get(dym, i), pnl_vect_c
    }
```

```
  derivs(bound + step, maturity, lambda, theta, eta, k, sigmamean, gamma1, r0, s
  for (i = 0; i < n; i++) pnl_vect_complex_set(yout, i, Cadd(pnl_vect_complex_ge
  pnl_vect_complex_free(&dym);
  pnl_vect_complex_free(&dyt);
  pnl_vect_complex_free(&yt);
}
void static ode_solution(dcomplex u, double maturity, double lambda, double thet
{
  int i, j;
  double x, h;
  PnlVectComplex *v, *dv;
  v = pnl_vect_complex_create_from_dcomplex(nvar, Complex(0.0, 0.0));
  dv = pnl_vect_complex_create_from_dcomplex(nvar, Complex(0.0, 0.0));
  for (i = 0; i < nvar; i++) pnl_vect_complex_set(v, i, pnl_vect_complex_get(ybo
  x = x1;
  h = (x2 - x1) / nstep;
  for (j = 1; j <= nstep; j++)
    {
      derivs(x, maturity, lambda, theta, eta, k, sigmamean, gamma1, r0, sigma0,
      ode_solution_step(u, maturity, lambda, theta, eta, r0, k, sigmamean, gamma
      if ((double)(x + h) == x) printf("Step size too small in routine");
      x += h;
      for (i = 0; i < nvar; i++) pnl_vect_complex_set(v, i, pnl_vect_complex_get
    }
  pnl_vect_complex_free(&v);
  pnl_vect_complex_free(&dv);
}


void static chf(dcomplex u, double kappa, double sigmamean, double gamma1, doubl
{

  dcomplex expon = Complex(0.0, 0.0);
  if (approximation_flag == 0)
    {
      double a1, a2, b1, b2, c1, c2, fc1, fc2, fd1, fd2, flambda1, flambda2, cla
      dcomplex D1, D2, G1, G2, A, Bx, Bsigma, Br;
      double tau;
      tau = maturity - 0.0;
      funcD1(u, lambda, eta, &D1);
      funcD2(u, gamma1, rho12, kappa, &D2);
```

```
    funcG1(lambda, D1, &G1);
    funcG2(u, kappa, gamma1, rho12, D2, &G2);
    funcC(1.0, kappa, gamma1, &fc1);
    funcC(1.0, lambda, eta, &fc2);
    funcLambda(1.0, kappa, sigma0, gamma1, &flambda1);
    funcLambda(1.0, lambda, r0, eta, &flambda2);
    funcD(kappa, sigmamean, gamma1, &fd1);
    funcD(lambda, theta, eta, &fd2);
    Lambda1(1.0, fc1, flambda1, fd1, &clambda1);
    Lambda1(1.0, fc2, flambda2, fd2, &clambda2);
    contA(sigmamean, kappa, gamma1, &a1);
    contA(theta, lambda, eta, &a2);
    contB(sigma0, a1, &b1);
    contB(r0, a2, &b2);
    contC(a1, b1, clambda1, &c1);
    contC(a2, b2, clambda2, &c2);
    funcBx(u, &Bx);
    funcBr(u, tau, lambda, eta, D1, G1, &Br);
    funcBsigma(u, tau, kappa, gamma1, rho12,  D2, G2, &Bsigma);
    funcA(u, tau, kappa, sigmamean, gamma1, lambda, theta, eta, rho12, rho13,

    expon = Cadd(A, RCmul(sigma0, Bsigma));
    expon = Cadd(expon, RCmul(r0, Br));
    *result = Cexp(expon);
  }
else
  {
    PnlVectComplex *ybound, *dybound, *yout, *vout;
    ybound = pnl_vect_complex_create_from_dcomplex(7, Complex(0.0, 0.0));
    dybound = pnl_vect_complex_create_from_dcomplex(7, Complex(0.0, 0.0));
    yout = pnl_vect_complex_create_from_dcomplex(7, Complex(0.0, 0.0));
    vout = pnl_vect_complex_create_from_dcomplex(7, Complex(0.0, 0.0));
    ode_bound(u, ybound, dybound);
    ode_solution(u, maturity, lambda, theta, eta, r0, kappa, sigmamean, gamma1

    expon = pnl_vect_complex_get(vout, 6);
    expon = Cadd(expon, RCmul(r0, pnl_vect_complex_get(vout, 1)));
    expon = Cadd(expon, RCmul(sigma0, pnl_vect_complex_get(vout, 4)));
    *result = Cexp(expon);
    pnl_vect_complex_free(&ybound);
    pnl_vect_complex_free(&dybound);
```

```
      pnl_vect_complex_free(&yout);
      pnl_vect_complex_free(&vout);
    }
}

//COSINE method to calculate inverse fourier integral
static  double cosine_function_xi(double d, double c, double dma, double cma, do
{


  double res = 1. / (1 + fnpi * fnpi) * ((cos(dma) + fnpi * sin(dma)) * exp(d) -
  return res;
}

static double cosine_function_psi(double d, double c, double dma, double cma, do
{

  double res = (fnpi == 0.) ? (d - c) : (sin(dma) - sin(cma)) / fnpi;
  return res;
}

static double cosine_Vk_call(double K, double a, double b, double fnpi)
{
  double ma = -a * fnpi;
  double bma = (b - a) * fnpi;
  return 2. / (b - a) * K * (cosine_function_xi(b, 0, bma, ma, fnpi) - cosine_fu
}
static double cosine_Vk_put(double K, double a, double b, double fnpi)
{

  double ma = -a * fnpi;
  return 2. / (b - a) * K * (-cosine_function_xi(0, a, ma, 0, fnpi) + cosine_fun
}
static double cosine_Vk(int callput_flag, double K, double a, double b, double f
{

  double result = 0.0;
  if (callput_flag == 0) result = cosine_Vk_call(K, a, b, fnpi);
  else result = cosine_Vk_put(K, a, b, fnpi);
  return result;
}
```

```
static void cosine_left_bound(double L, double S0, double K, double sigmamean, d
{

  double c1 = 0.0, c2 = 0.0;
  c1 = r0 * tau + (1. - exp(-kappa * tau)) * (sigmamean - sigma0) / (2.0 * kappa
  c2 = (1.0 / (8.0 * pow(kappa, 3))) * (gamma1 * tau * kappa * exp(-kappa * tau)

  /*Truncation range*/
  *a = c1 - L * pow(fabs(c2), 0.5) + log(S0 / K);
  *b = c1 + L * pow(fabs(c2), 0.5) + log(S0 / K);
}

static void HCIR_COSINE(double kappa, double sigmamean, double gamma1, double si
{

  double sum = 0, sumd = 0, a = 0, b = 0;
  double x = log(S0 / K);
  double invbma = 0.0;
  double xma = 0.0;
  dcomplex fnpi = CZERO;
  dcomplex phi = CZERO;
  int i;
  cosine_left_bound(L, S0, K, sigmamean, sigma0, eta, gamma1, kappa, r0, maturit
  xma = x - a;
  invbma = M_PI / (b - a);
  chf(fnpi, kappa, sigmamean, gamma1, sigma0, lambda, theta, eta, r0, rho12, rho
  sum = 0.5 * phi.r * cosine_Vk(callput_flag, K, a, b, fnpi.r);
  sumd = 0.;
  for (i = 1; i < N; i++)
    {
      fnpi.r += invbma;
      chf(fnpi, kappa, sigmamean, gamma1, sigma0, lambda, theta, eta, r0, rho12,
      sum += (phi.r * cos(fnpi.r * xma) - phi.i * sin(fnpi.r * xma)) * cosine_Vk
      sumd -= (phi.i * cos(fnpi.r * xma) + phi.r * sin(fnpi.r * xma)) * fnpi.r *
    }
  *GreekDelta = sumd / S0;
  *Price = sum;
}

int ApGrzelakOosterlee(double spot, NumFunc_1  *p, double maturity, double r0, d
{
```

```
  double K, price, delta;
  int N, L;
  int flag_call;


  //ATTENTION Correlation rv int his algorithm = 0

  N = 100; // discrect steps
  L = 10; //parameter for integration bound

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

  if ((p->Compute) == &Call)
    flag_call = 0;
  else
    flag_call = 1;

  HCIR_COSINE(kv, vbar, sigmav, v0, kr, rbar, sigmar, r0, rho12, rho13, K, spot,

  //Price
  *ptprice = price;


  //Delta
  *ptdelta = delta;

  return OK;
}

int CALC(AP_GrzelakOosterlee)(void *Opt, void *Mod, PricingMethod *Met)
{
  TYPEOPT *ptOpt = (TYPEOPT *)Opt;
  TYPEMOD *ptMod = (TYPEMOD *)Mod;

  return ApGrzelakOosterlee(ptMod->S0.Val.V_PDOUBLE,
                            ptOpt->PayOff.Val.V_NUMFUNC_1,
                            ptOpt->Maturity.Val.V_DATE - ptMod->T.Val.V_DATE,
                            ptMod->r0.Val.V_PDOUBLE
                            , ptMod->kr.Val.V_PDOUBLE,
                            ptMod->thetar.Val.V_PDOUBLE,
                            ptMod->Sigmar.Val.V_PDOUBLE,
```

```
                                ptMod->V0.Val.V_PDOUBLE
                                , ptMod->kV.Val.V_PDOUBLE,
                                ptMod->thetaV.Val.V_PDOUBLE,
                                ptMod->SigmaV.Val.V_PDOUBLE,
                                ptMod->RhoSr.Val.V_PDOUBLE,
                                ptMod->RhoSV.Val.V_PDOUBLE,
                                Met->Par[0].Val.V_ENUM.value,
                                &(Met->Res[0].Val.V_DOUBLE),
                                &(Met->Res[1].Val.V_DOUBLE));
}

static int CHK_OPT(AP_GrzelakOosterlee)(void *Opt, void *Mod)
{
  if ((strcmp(((Option *)Opt)->Name, "PutEuro") == 0) || (strcmp(((Option *)Opt)
    return OK;
  return WRONG;
}
#endif //PremiaCurrentVersion
static PremiaEnumMember ApproximationMembers[] =
{
  { "Deterministic approximation", 0 },
  { "Stochastic approximation", 1 },
  { NULL, NULLINT }
};

static DEFINE_ENUM(Approximation, ApproximationMembers);
static int MET(Init)(PricingMethod *Met, Option *Opt)
{
  if (Met->init == 0)
    {
      Met->init = 1;
      Met->HelpFilenameHint = "ap_grzelak_oosterlee";
      Met->Par[0].Val.V_ENUM.value = 0;
      Met->Par[0].Val.V_ENUM.members = &Approximation;
    }

  return OK;
}

PricingMethod MET(AP_GrzelakOosterlee) =
{
```

```
  "AP_GrzelakOosterlee",
  { {"Approximation method (Rho r V=0)", ENUM, {100}, ALLOW},
    {" ", PREMIA_NULLTYPE, {0}, FORBID}
  },
  CALC(AP_GrzelakOosterlee),
  { {"Price", DOUBLE, {100}, FORBID},
    {"Delta", DOUBLE, {100}, FORBID} ,
    {" ", PREMIA_NULLTYPE, {0}, FORBID}
  },
  CHK_OPT(AP_GrzelakOosterlee),
  CHK_ok,
  MET(Init)
};
```