Help

```
#include <stdlib.h>
#include  "
href../../../mod/hk1d/hk1d_stdi/hk1d_stdi_h_src.pdfhk1d_stdi.h"
#include "
href../../../mod/hk1d/hk1d_stdi/mathsb_h_src.pdfmathsb.h"
#include "
href../../../mod/hk1d/hk1d_stdi/currentzcb_h_src.pdfcurrentzcb.h"
#include "
href../../../mod/hk1d/hk1d_stdi/hktree_h_src.pdfhktree.h"

/*Swaption=Option on Coupon-Bearing Bond*/
/*All details comments for the functions used here are mainly in "hwtree1dinclud

static void HK_iterations(int flat_flag, double r_flat, char *init, double a, do
                          double T0, double per, int m, double K0, int xnumber,

////////////////////////////////////////////////////////////////////////////////
// computes V_0(K), the current Hull-White-price of the digital (T,S)-caplet wit
////////////////////////////////////////////////////////////////////////////////
/*static double HW_DigitalCaplet(double a0, double sigma0, double T, double S, d
  {
  double sigma_P, log_term;

  sigma_P = sigma0 * (exp(-a0*T) - exp(-a0*S))/a0 * sqrt( (exp(2*a0*T)-1)/(2*a0)

  log_term = log( P0T / P0S / (tau0*K+1) );


  return tau0 * P0S * cdf_nor( log_term/sigma_P - sigma_P/2 );
  }*/


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

```
  return AVAILABLE_IN_FULL_PREMIA;
}
#else


////////////////////////////////////////////////////////////////////////////
// solves V_0(K)=x, where V_0(K) is the current Hull-White-price of the digital
////////////////////////////////////////////////////////////////////////////
static double Inv_HW_DigitalCaplet(double a0, double sigma0, double T, double S,
{
  double tau0_inv, sigma_P, exp_term, K;

  tau0_inv = 1 / tau0;

  sigma_P = sigma0 * (exp(-a0 * T) - exp(-a0 * S)) / a0 * sqrt((exp(2 * a0 * T)

  exp_term = exp(-SQR(sigma_P) / 2 - sigma_P * pnl_inv_cdfnor(x / (tau0 * P0S)))

  K = tau0_inv * P0T / P0S * exp_term - tau0_inv;
  /*if (K<0) printf("Inv_HW_DigitalCaplet returns the negative strike K = %f\ n"

  return K;
}

////////////////////////////////////////////////////////////////////////////
// solves V_0^{m-1,HK}(K) = V_0^{m-1,HW}(K) in terms of sigma_HK, where V_0^{m-1
// is the current Hunt-Kennedy-price resp. Hull-White price of the digital (T_{m
////////////////////////////////////////////////////////////////////////////
/*static double ComputeSigma_HK(double a, double sigma_HW, double Tm_1, double T
  {
  double sigma_HK, R0m_1, p, q, sigma_P;

  R0m_1 = (P0Tm_1/P0Tm - 1.) / tau;
  sigma_P = sigma_HW * (exp(-a*Tm_1) - exp(-a*Tm))/a * sqrt( (exp(2.*a*Tm_1)-1)/
  p   = 2. * log( (R0m_1+tau)/(K+tau) ) / sigma_P  -  sigma_P;
  q = -2. * log( R0m_1/K );
  sigma_HK = (sqrt( SQR(p)/4. - q ) - p/2.) / sqrt( (exp(2.*a*Tm_1)-1)/(2.*a) );

  return sigma_HK;
  }*/
```

```
////////////////////////////////////////////////////////////////////////////
// functional form of the INVERSE of the numeraire at T[m-1], i.e. of 1/P(T_{m-1
////////////////////////////////////////////////////////////////////////////
static double N_mminus1(double x, double C_2)
{
  return 1 + C_2 * exp(x);
}

////////////////////////////////////////////////////////////////////////////
// functional form of the INVERSE of the numeraire at T[m-2], i.e. of 1/P(T_{m-2
////////////////////////////////////////////////////////////////////////////
/*static double N_mminus2(double a_HW, double sigma_HW, double T_mminus2, double
  double P0T_mminus1, double P0T_m, double C_0, double C_1, double Sig, double x
  {
  double result, J_term, P_term, V0_market_inv;

  P_term = 1 + C_0 * exp(x);
  J_term = P0T_m * tau_mminus2 * ( cdf_nor(-x/Sig) + C_1*cdf_nor(-x/Sig+Sig) );
  V0_market_inv =  Inv_HW_DigitalCaplet( a_HW, sigma_HW, T_mminus2, T_mminus1, t

  result = P_term * ( 1 + tau_mminus2 * V0_market_inv);

  return result;
  }*/




////////////////////////////////////////////////////////////
// returns the variance of the HK-process x_t given x_s
////////////////////////////////////////////////////////////
static double SigmaSqr(double t, double s, double sigma, double a)
{
  return SQR(sigma) * (exp(2.*a * t) - exp(2.*a * s)) / (2 * a);
}
```

```
////////////////////////////////////////////////////////////////
// returns (U_{t,s}f)(x), where U is the semigroup of operators
// corresponding to the HK-process
////////////////////////////////////////////////////////////////
static double U(double t, double s, discrete_fct *f, double x, double sigma, dou
{
  return NormalTab(x, SigmaSqr(t, s, sigma, a), f);
}



static void SetUf(discrete_fct *g, double t, double s, discrete_fct *f, double s
// Sets g = U_{t,s}f in a reasonable way
{
  SetNf(g, SigmaSqr(t, s, sigma, a), f);
}



static double UfUpBound(discrete_fct *f, double t, double s, double vmax, double
// returns the minimum of all x>=f.xleft such that U_{t,s}(f*1_{(x,infty)})(0) <
{
  return NfUpBound(f,  SigmaSqr(t, s, sigma, a), vmax);
}


static double UfLoBound(discrete_fct *f, double t, double s, double vmin, double
// returns the maximum of all x<f.right such that U_{t,s}(f*1_{(x,infty)})(0) >
{
  return NfLoBound(f,  SigmaSqr(t, s, sigma, a), vmin);
}


void HK_iterations(int flat_flag, double r_flat, char *init, double a, double si
                   double T0, double per, int m, double K0, int xnumber, discret
// flat_flag           : flag to decide wether initial yield curve is flat at r_
// a, sigma_HW         : parameters of the HW-model representing the market ("a"
// sigma_HK            : parameter of the HK-process
// T0                  : first HK-date
```

```
// per                  : HK-periodicity
// m                    : number of HK-dates
// K0                   : calibration strike (for the computation of sigma_HK)
// N                    : functional forms of the INVERSE of the numeraire at T[i
// xnumber              : parameter for the discretization of the functional form
{

  double *T;                    /* HK-dates */
  double *tau;                  /* tau[i] = year fraction from T[i] to T[i+1] *
  double *P0;                   /* P0[i] = P(0,T[i]) (initial zcb prices) */
  double **Sigma;               /* corresponds to Sigma_{T[i],T[j-1]}, where T[
  /* here SQR(Sigma_{t,s}) is the variance of x_t given x_s */
  double C_2;                   /* corresponds to the constant C_2 in the formu


  double x, s, result, J_term, P_term, V0_market_inv;
  double xle, xste, xri, eps;
  double xleft, xstep;     /* parameters for the discretization of the functiona
  int i, j;
  discrete_fct Ptilde_i, Ptilde_ix;


  ////////////////////////////////////////
  // initialisation of the main variables //


  // HK-dates
  T = malloc((m + 1) * sizeof(double));
  for (i = 0; i <= m; i++)  T[i] = i * per + T0;


  tau = malloc(m * sizeof(double));
  for (i = 0; i < m; i++)  tau[i] = per;


  P0 = malloc((m + 1) * sizeof(double));
  for (i = 0; i <= m; i++)  P0[i] = CurrentZCB(T[i], flat_flag, r_flat, init);


  Sigma = malloc(m * sizeof(double *));
  for (i = 0; i < m; i++)
```

```
  {
    Sigma[i] = malloc((i + 1) * sizeof(double));
    for (j = 0; j <= i; j++)
      {
        if (j == 0) s = 0;
        else s = T[j - 1];
        Sigma[i][j] = sigma_HK * sqrt((exp(2 * a * T[i]) - exp(2 * a * s)) / (
      }
  }


// constant in the formula for N[m-1]
C_2 = (P0[m - 1] / P0[m] - 1) * exp(-SQR(Sigma[m - 1][0]) / 2);


////////////////////////////////////////////////////////////////////////
// initialization of N[m-1] (for which we have an explicit formula !) //
////////////////////////////////////////////////////////////////////////
xleft = MAX(-SQR(Sigma[m - 1][0]) - Sigma[m - 1][0] * sqrt(40.), -10.);
xstep = 2 * fabs(xleft) / (double)xnumber;
Set_discrete_fct(&N[m - 1], xleft, xstep, xnumber);
for (j = 0; j < N[m - 1].xnumber; j++)
  {
    x = N[m - 1].xleft + j * N[m - 1].xstep;
    N[m - 1].val[j] =  N_mminus1(x , C_2);
  }

////////////////////////////////////////////////
// iterative computation of the N[m-2],...,N[0] //
////////////////////////////////////////////////
for (i = m - 2; i >= 0; i--)
  {

    ////////////////////////////////////////////////
    // setting of P^tilde_i := U_{T[i+1],T[i]} N[i+1] //
    ////////////////////////////////////////////////
    SetUf(&Ptilde_i, T[i + 1], T[i], &N[i + 1], sigma_HK, a);
    // sets Ptilde_i such that domain( Ptilde_i ) = [ U_{T[i+1],T[i]} N[i+1] >

    ////////////////////
    // setting of N[i] //
```

```
//////////////////////
eps = 0.000001;
xle = UfUpBound(&Ptilde_i, T[i], 0., P0[i + 1] / P0[m] - eps, sigma_HK, a)
xri = UfLoBound(&Ptilde_i, T[i], 0., eps, sigma_HK, a);
xste = (xri - xle) / (double)(xnumber - 1);
Set_discrete_fct(&N[i], xle, xste, xnumber);

////////////////////////////////////////////////////////////////////////
// initialization of P^tilde_{i,x} as a (restricted) copy of P^tilde_i //
////////////////////////////////////////////////////////////////////////
Set_discrete_fct(&Ptilde_ix, N[i].xleft, N[i].xstep, N[i].xnumber + 1);
for (j = 0; j < Ptilde_ix.xnumber; j++)
  {
    x = Ptilde_ix.xleft + j * Ptilde_ix.xstep;
    Ptilde_ix.val[j] =  U(T[i + 1], T[i], &N[i + 1], x, sigma_HK, a);
  }



////////////////////////////////////////////////////////////////////////
// evaluation of N_i in its discretizing points N[i].xleft + j*N[i].xstep
////////////////////////////////////////////////////////////////////////
for (j = 0; j < N[i].xnumber; j++)
  {
    x = N[i].xleft + j * N[i].xstep; // observe: x = Ptilde_ix.xleft + j*P
    P_term =  Ptilde_ix.val[j];      // hence: P_term = P_i^tilde(x) !!!

    Ptilde_ix.val[j] = 0;
    // VERY IMPORTANT: now Ptilde_ix corresponds really to P^tilde_{i,x} :

    J_term = P0[m] * tau[i] * U(T[i], 0., &Ptilde_ix, 0., sigma_HK, a);

    V0_market_inv =  Inv_HW_DigitalCaplet(a, sigma_HW, T[i], T[i + 1], tau

    result = P_term * (1 + tau[i] * V0_market_inv);
    // now we have:  result = N_i(x)

    N[i].val[j] = result;
  }

Delete_discrete_fct(&Ptilde_i);
```

```
      Delete_discrete_fct(&Ptilde_ix);

    } // end of i-loop



  ////////////////////////////////
  // free the variables         //

  free(T);
  free(tau);
  free(P0);
  for (i = 0; i < m; i++) free(Sigma[i]);
  free(Sigma);

  // end of: free the variables //
  ////////////////////////////////
}


////////////////////////////////////////////////////////////////////////////
// prices the bermudan call/put on P(T0,S0) via a trinomial tree for the HK-proc
//   n=1 (European case): exercise date = T0
//                        uses the HK-dates T[i]=i*per+T0 for i=0,...,m; here pe
//   n>1                : exercise dates = 0,per,2*per,...,(n-1)*per=T0; here pe
//                        uses the HK-dates T[i]=i*per for i=0,...,m; here m=S0/
////////////////////////////////////////////////////////////////////////////
static int zbo_hk1d(int flat_flag, double a, double t0, double sigma_HW, double
{
  // flat_flag          : flag to decide wether initial yield curve is flat at
  // a, sigma_HW        : parameters of the HW-model representing the market ("
  // T0                 : maturity of the option
  // S0                 : maturity of the ZCB underlying the option
  // call               : call (1) or put (0)
  // n                  : number of exercise dates of the option
  // t0                 : time for which the price of the call is computed
  // K                  : strike of the call
  // N_step             : number of time steps in the tree for the HK-process
  // xnumber            : parameter for the discretization of the functional fo

  double *T;                    /* HK-dates */
```

```
discrete_fct *N;               /* functional forms of the INVERSE of the numer
/* i.e. of 1/P(T_i,T_m), for i=0,...,m-1 */
struct Tree Tr;                /* tree for the HK-process */


double x, **disc_payoff, first_ex_date, Nix, disc_price;
double per;                              /* period. of the HK-dates    */
int m;                                   /* number of HK-dates         */
double sigma_HK;                         /* parameter of the HK-proces */
int i, j, *ind, *Size;
int call;
double K;

K = p->Par[0].Val.V_DOUBLE;
if ((p->Compute) == &Call)
  call = 1;
else
  /*if ((p->Compute)==&Put)*/
  call = 0;

if (am == 0)
  n = 1;

////////////////////////////////////////
// initialisation of the main variables //


// European or not
if (n == 1)
  {
    m = 5;
    per = (S0 - T0) / (double)m;
    first_ex_date = T0;
  }
else
  {
    per = T0 / (double)(n - 1);
    m = S0 / per;
    first_ex_date = 0.;
  }
```

```
// HK-dates
T = malloc((m + 1) * sizeof(double));
for (i = 0; i <= m; i++)  T[i] = i * per + first_ex_date;
// Observe:  T[n-1]=T0  and  T[m]=S0  !!


// choice of sigma_HK
sigma_HK = sigma_HW;


// functional forms of the INVERSE of the numeraire at T[i], i.e. of 1/P(T_i,T
N = malloc(m * sizeof(discrete_fct));

// end of: initialisation of the main variables //
////////////////////////////////////////////////


////////////////////////////////////////////////////////////////
// construction of a trinomial tree for the HK-process         //


// the maturity T0=T[n-1] of the option is the final time of the tree, N_step
SetTimegrid(&Tr, T[n - 1], N_step);


// construct a tree for the HK-process (x_t) given by: dx_t = sigma*exp(a*t) d
SetHKtree(&Tr, a, sigma_HK);


// end of: construction of a trinomial tree for the HK-process //
////////////////////////////////////////////////////////////////


ind = malloc(n * sizeof(int));
for (i = 0; i < n; i++)
  ind[i] = indiceTime(&Tr, T[i]);
// we have: Tr.t[ ind[i] ]  = T[i]


Size = malloc(n * sizeof(int));
```

```
for (i = 0; i < n; i++)
  Size[i] = Tr.TSize[ind[i]];
// at T[i], the tree has Size[i] nodes


// Construct the functional forms N[0],...,N[m-1]
HK_iterations(flat_flag, r_flat, init, a, sigma_HW, sigma_HK,
              T[0], per, m, K, xnumber, N);


disc_payoff = malloc(n * sizeof(double *));
// disc_payoff[i] will represent the discounted payoff of the approx. bermudan
for (i = 0; i < n; i++)
  {
    disc_payoff[i] = (double *)calloc(Size[i], sizeof(double));

    for (j = 0; j < Size[i]; j++)
      {
        x = Tr.pLRij[ ind[i] ][j];
        Nix = InterpolDiscreteFct(&N[i], x);  // corresponds to 1/P(T[i],T[m])
        if (Nix > 0)
          {
            if (call)
              {
                if (1 / Nix > K)  disc_payoff[i][j] = Nix * (1 / Nix - K);
              }
            else
              {
                if (K > 1 / Nix)  disc_payoff[i][j] = Nix * (K - 1 / Nix);
              }
          }
      }
  }
// now disc_payoff[i] represents  1/P(T[i],T[m]) * ( P(T[i],T[m]) - K )_+
// respectively 1/P(T[i],T[m]) * ( K - P(T[i],T[m]) )_+
// which is the correct discounted payoff at T[i] of the bermudan option !!


initPayoff1(&Tr, T[n - 1]);
for (i = 0; i < n; i++)
  {
```

```
    for (j = 0; j < Size[i]; j++)
      {
        Tr.Payoffunc[ind[i]][j] = disc_payoff[i][j];
      }
  }


// Compute the bermudan Call ZCB from the last exercise date T[n-1] to 0 in Tr
Computepayoff1(&Tr, T[n - 1]);


// return  plQij[0][1]  as discounted price of the CallZCB
if (t0 == 0)
  {
    disc_price = Tr.pLQij[0][1];
    *price = CurrentZCB(T[m], flat_flag, r_flat, init) * disc_price;
  }
else printf("Evaluation in t>0 is not implemented.\ n");


////////////////////////////////
// free the variables         //

free(T);
for (i = 0; i < m; i++)  Delete_discrete_fct(&N[i]);
free(N);

DeletePayoff1(&Tr, T0);
DeleteTree(&Tr);

for (i = 0; i < n; i++)  free(disc_payoff[i]);
free(disc_payoff);
free(ind);
free(Size);

// end of: free the variables //
////////////////////////////////

return OK;

}
```

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

  return zbo_hk1d(ptMod->flat_flag.Val.V_INT,
                  ptMod->a.Val.V_DOUBLE,
                  ptMod->T.Val.V_DATE,
                  ptMod->Sigma.Val.V_PDOUBLE,
                  MOD(GetYield)(ptMod),
                  MOD(GetCurve)(ptMod),
                  ptOpt->BMaturity.Val.V_DATE,
                  ptOpt->OMaturity.Val.V_DATE,
                  ptOpt->PayOff.Val.V_NUMFUNC_1,

                  ptOpt->EuOrAm.Val.V_BOOL,
                  Met->Par[0].Val.V_LONG,
                  Met->Par[1].Val.V_INT,
                  Met->Par[2].Val.V_INT,
                  &(Met->Res[0].Val.V_DOUBLE));
}


static int CHK_OPT(TR_ZBO)(void *Opt, void *Mod)
{

  if ((strcmp(((Option *)Opt)->Name, "ZeroCouponCallBondEuro") == 0) || (strcmp(
    return OK;
  else
    return WRONG;
}

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

      Met->Par[0].Val.V_LONG = 140;
```

```
      Met->Par[1].Val.V_INT = 1000;
      Met->Par[2].Val.V_INT = 10;


    }
  return OK;
}

PricingMethod MET(TR_ZBO) =
{
  "TR_HK1d_ZBO",
  { {"TimeStepNumber", LONG, {100}, ALLOW},
    {"Parameter for the discretization of the functional forms ", INT, {100}, AL
    {"Number of exercise dates : only in the American case", INT, {100}, ALLOW},
    {" ", PREMIA_NULLTYPE, {0}, FORBID}
  },
  CALC(TR_ZBO),
  {{"Price", DOUBLE, {100}, FORBID}/*,{"Delta",DOUBLE,{100},FORBID}*/ , {" ", PR
  CHK_OPT(TR_ZBO),
  CHK_ok,
  MET(Init)
} ;
```