Help

```
#include  "
href../../../mod/hes1d/hes1d_stda/hes1d_stda_h_src.pdfhes1d_stda.h"
#include "
href../../../common/enums_h_src.pdfenums.h"
#include "
href../../../common/error_msg_h_src.pdferror_msg.h"

#if defined(PremiaCurrentVersion) && PremiaCurrentVersion < (2015+2) //The "#els
static int CHK_OPT(FD_HybridTree_GLWB_HES)(void *Opt, void *Mod)
{
  return NONACTIVE;
}
int CALC(FD_HybridTree_GLWB_HES)(void *Opt, void *Mod, PricingMethod *Met)
{
  return AVAILABLE_IN_FULL_PREMIA;
}
#else

/*Vettori*/
static char *infilename;
static double gamma_i;
static int ratchet;
static double tt, s0, r, divid, v0, theta, kappa, omega, rho, theta_sc;
static double alpha_m;
static int Nt, N;

static double **V, * *P_old, * *P_new;
static double **f;
static int **f_down, * *f_up;
static double **pu_f, * *pd_f;

static double lm_insurance[500];
static double lm_age[500];
static double Rt[500];
static double Mt[500];
static double Kt[500];
static double Bt[500];
static double Gt[500];
static int n_step_for_period;
```

```
#define MAXIT 30

static double rtsec(double (*func)(double), double x1, double x2, double xacc)
{
  int j;
  double fl, f, dx, swap, xl, rts;

  fl = (*func)(x1);
  f = (*func)(x2);
  if (fabs(fl) < fabs(f))
    {
      rts = x1;
      xl = x2;
      swap = fl;
      fl = f;
      f = swap;
    }
  else
    {
      xl = x1;
      rts = x2;
    }
  for (j = 1; j <= MAXIT; j++)
    {
      dx = (xl - rts) * f / (f - fl);
      xl = rts;
      fl = f;
      rts += dx;
      f = (*func)(rts);
      if (fabs(dx) < xacc || f == 0.0) return rts;
    }
  printf("Maximum number of iterations exceeded in rtsec\ n");
  return 0.0;
}
#undef MAXIT



/*Memory Allocation*/
static int memory_allocation(int Nt, int N)
```

```
{
  int i;

  V = (double **)calloc(Nt + 1, sizeof(double *));
  if (V == NULL)
    return MEMORY_ALLOCATION_FAILURE;
  for (i = 0; i < Nt + 1; i++)
    {
      V[i] = (double *)calloc(Nt + 1, sizeof(double));
      if (V[i] == NULL)
        return MEMORY_ALLOCATION_FAILURE;
    }


  pu_f = (double **)calloc(Nt + 1, sizeof(double *));
  if (pu_f == NULL)
    return MEMORY_ALLOCATION_FAILURE;
  for (i = 0; i < Nt + 1; i++)
    {
      pu_f[i] = (double *)calloc(Nt + 1, sizeof(double));
      if (pu_f[i] == NULL)
        return MEMORY_ALLOCATION_FAILURE;
    }

  pd_f = (double **)calloc(Nt + 1, sizeof(double *));
  if (pd_f == NULL)
    return MEMORY_ALLOCATION_FAILURE;
  for (i = 0; i < Nt + 1; i++)
    {
      pd_f[i] = (double *)calloc(Nt + 1, sizeof(double));
      if (pd_f[i] == NULL)
        return MEMORY_ALLOCATION_FAILURE;
    }


  f = (double **)calloc(Nt + 1, sizeof(double *));
  if (f == NULL)
    return MEMORY_ALLOCATION_FAILURE;
  for (i = 0; i < Nt + 1; i++)
    {
      f[i] = (double *)calloc(Nt + 1, sizeof(double));
```

```
      if (f[i] == NULL)
        return MEMORY_ALLOCATION_FAILURE;
    }

  f_down = (int **)calloc(Nt + 1, sizeof(int *));
  if (f_down == NULL)
    return MEMORY_ALLOCATION_FAILURE;
  for (i = 0; i < Nt + 1; i++)
    {
      f_down[i] = (int *)calloc(Nt + 1, sizeof(int));
      if (f_down[i] == NULL)
        return MEMORY_ALLOCATION_FAILURE;
    }

  f_up = (int **)calloc(Nt + 1, sizeof(int *));
  if (f_up == NULL)
    return MEMORY_ALLOCATION_FAILURE;
  for (i = 0; i < Nt + 1; i++)
    {
      f_up[i] = (int *)calloc(Nt + 1, sizeof(int));
      if (f_up[i] == NULL)
        return MEMORY_ALLOCATION_FAILURE;
    }

  P_old = (double **)malloc((N + 1) * sizeof(double *));
  for (i = 0; i <= N; i++)
    P_old[i] = (double *)malloc((Nt + 1) * sizeof(double));

  P_new = (double **)malloc((N + 1) * sizeof(double *));
  for (i = 0; i <= N; i++)
    P_new[i] = (double *)malloc((Nt + 1) * sizeof(double));

  return OK;
}


static void free_memory(int Nt, int N)
{
  int i;

  for (i = 0; i < Nt + 1; i++)
```

```
    free(V[i]);
  free(V);

  for (i = 0; i < Nt + 1; i++)
    free(pu_f[i]);
  free(pu_f);

  for (i = 0; i < Nt + 1; i++)
    free(pd_f[i]);
  free(pd_f);

  for (i = 0; i < Nt + 1; i++)
    free(f[i]);
  free(f);

  for (i = 0; i < Nt + 1; i++)
    free(f_up[i]);
  free(f_up);

  for (i = 0; i < Nt + 1; i++)
    free(f_down[i]);
  free(f_down);

  for (i = 0; i < N + 1; i++)
    free(P_old[i]);
  free(P_old);

  for (i = 0; i < N + 1; i++)
    free(P_new[i]);
  free(P_new);

  return;
}

static double compute_f(double r, double omega)
{
  return 2.*sqrt(r) / omega;
}

static double compute_v(double R, double omega)
{
```

```
  double val;

  val = SQR(R) * SQR(omega) / 4.;
  if (R > 0.)
    val = SQR(R) * SQR(omega) / 4.;
  else
    val = 0.0;
  return val;
}

static double compute_S(double Y, double rv, double omega, double rho)
{
  double val;

  val = exp(Y) * exp(rho * rv / omega);

  return val;
}

/*Calibration of the tree  the stochastic volatilty v*/
static void tree_v(double tt, double v0, double kappa, double theta, double omeg
{
  int i, j;
  int z;
  double Ru, Rd;
  double mu_r, v_curr;
  double dt, sqrt_dt;

  /*Fixed tree for R=f*/
  f[0][0] = compute_f(v0, omega);

  dt = tt / (double)Nt;
  sqrt_dt = sqrt(dt);

  V[0][0] = compute_v(f[0][0], omega);
  f[1][0] = f[0][0] - sqrt_dt;
  f[1][1] = f[0][0] + sqrt_dt;
  V[1][0] = compute_v(f[1][0], omega);
  V[1][1] = compute_v(f[1][1], omega);
  for (i = 1; i < Nt; i++)
    for (j = 0; j <= i; j++)
```

```
    {
      f[i + 1][j] = f[i][j] - sqrt_dt;
      f[i + 1][j + 1] = f[i][j] + sqrt_dt;
      V[i + 1][j] = compute_v(f[i + 1][j], omega);
      V[i + 1][j + 1] = compute_v(f[i + 1][j + 1], omega);
    }

/*Evolve tree for f*/
for (i = 0; i < Nt; i++)
  {
    for (j = 0; j <= i; j++)
      {
        /*Compute mu_f*/
        v_curr = V[i][j];

        mu_r = kappa * (theta - v_curr);

        z = 0;
        while ((V[i][j] + mu_r * dt < V[i + 1][j - z])
               && (j - z >= 0))
          {

            z = z + 1;
          }
        f_down[i][j] = -z;
        Rd = V[i + 1][j - z];

        if (z > 0)
          z = 0;
        else z = 1;
        while ((V[i][j] + mu_r * dt > V[i + 1][j + z])
               && (j + z <= i))
          {
            z = z + 1;
          }

        Ru = V[i + 1][j + z];

        f_up[i][j] = z;
        pu_f[i][j] = (V[i][j] + mu_r * dt - Rd) / (Ru - Rd);
```

```
          if ((Ru - 1.e-9 > V[i + 1][i + 1]) || (j + f_up[i][j] > i + 1))
            {
              pu_f[i][j] = 1;

              f_up[i][j] = i + 1 - j;
              f_down[i][j] = i - j;
            }

          if ((Rd + 1.e-9 < V[i + 1][0]) || (j + f_down[i][j] < 0))
            {
              pu_f[i][j] = 0.;
              f_up[i][j] = 1 - j;
              f_down[i][j] = 0 - j;
            }
          pd_f[i][j] = 1. - pu_f[i][j];

        }
    }
}

/*Compute Price Bond*/
static double compute_price(double alpha_g)
{
  int i, j, k;
  double stock;
  int fv_up, fv_down;
  double l;
  double alpha, beta, gamma, alpha1, beta1, gamma1;
  int PriceIndex;
  double dx;
  double  *A, *B, *C, *A1, *B1, *C1, *Price, *S, *vect_y;
  double dt;
  double z, vv;
  double bound1, bound2;
  int Index;
  double A_IN, new_A, new_P = 0.;
  int anno, flag_monit, i_s;
  double s_value, s_value1;
  int current_mt_year, current_mt_year1;
  double log_s0;
  double val, val1, val_s, val_sim;
```

```
double price;
double discount;
double sigma=0.5;
double PRECISION_FDH=1.0e-5;

A_IN = s0;

A = (double *)malloc((N + 1) * sizeof(double));
B = (double *)malloc((N + 1) * sizeof(double));
C = (double *)malloc((N + 1) * sizeof(double));
A1 = (double *)malloc((N + 1) * sizeof(double));
B1 = (double *)malloc((N + 1) * sizeof(double));
C1 = (double *)malloc((N + 1) * sizeof(double));

vect_y = (double *)malloc((N + 1) * sizeof(double));
Price = (double *)malloc((N + 1) * sizeof(double));
S = (double *)malloc((N + 1) * sizeof(double));

dt = tt / (double)Nt;

 l=sigma*sqrt(tt)*sqrt(log(1.0/PRECISION_FDH))+fabs((r-divid-0.5*sigma)*tt);
dx = 2.0 * l / (double)N;
log_s0 = log(s0) - rho / omega * V[0][0];

for (j = 0; j <= N; j++)
  {
    vect_y[j] = log_s0 - l + (double)j * dx;
  }

/*Maturity conditions*/
for (k = 0; k <= Nt; k++)
  for (j = 0; j <= N; j++)
    {
      stock = compute_S(vect_y[j], V[Nt][k], omega, rho);
      P_old[j][k] = stock * Rt[56];
      P_old[j][k] = 0.;
    }
bound1 = 0.;
bound2 = 0.;

discount = exp(-r * dt);
```

```
anno = (int)tt;

/*Dynamic Programming*/
for (i = Nt - 1; i >= 0; i--)
  {

    current_mt_year1 = (int)((i + 1) * dt);
    current_mt_year = (int)((i) * dt);

    if ((((i) % n_step_for_period) == 0) && ((i) != Nt) && (i > 0))
      {
        flag_monit = 1;
        anno = anno - 1;
      }
    else flag_monit = 0;

    for (k = 0; k <= i; k++)
      {
        z = r - divid - alpha_g - alpha_m - 0.5 * V[i][k] - rho * kappa * (the
        vv = 0.5 * V[i][k] * (1. - SQR(rho));

        //Fully Implicit Scheme

        /*Lhs Factor of the fully implicit scheme*/
        alpha = theta_sc * (-vv * dt / SQR(dx) + z * dt / (2.*dx));
        beta = 1 + theta_sc * vv * 2 * dt / SQR(dx);
        gamma = theta_sc * (-vv * dt / SQR(dx) - z * dt / (2 * dx));

        for (PriceIndex = 1; PriceIndex <= N - 1; PriceIndex++)
          {
            A[PriceIndex] = alpha;
            B[PriceIndex] = beta;
            C[PriceIndex] = gamma;
          }
        B[1] = beta + alpha;
        B[N - 1] = beta + gamma;

        /*Rhs Factors*/
        alpha1 = (1. - theta_sc) * (vv * dt / SQR(dx) - z * dt / (2.*dx));
        beta1 = 1 - (1. - theta_sc) * vv * 2 * dt / SQR(dx);
        gamma1 = (1. - theta_sc) * (vv * dt / SQR(dx) + z * dt / (2 * dx));
```

```
for (PriceIndex = 1; PriceIndex <= N - 1; PriceIndex++)
  {
    A1[PriceIndex] = alpha1;
    B1[PriceIndex] = beta1;
    C1[PriceIndex] = gamma1;
  }

B1[1] = beta1 + alpha1;
B1[N - 1] = beta1 + gamma1;


/*Set Gauss*/
for (PriceIndex = N - 2; PriceIndex >= 1; PriceIndex--)
  B[PriceIndex] = B[PriceIndex] - C[PriceIndex] * A[PriceIndex + 1] /
for (PriceIndex = 1; PriceIndex < N; PriceIndex++)
  A[PriceIndex] = A[PriceIndex] / B[PriceIndex];
for (PriceIndex = 1; PriceIndex < N - 1; PriceIndex++)
  C[PriceIndex] = C[PriceIndex] / B[PriceIndex + 1];

fv_up = f_up[i][k];
fv_down = f_down[i][k];

//F_U

//Initialise
for (PriceIndex = 1; PriceIndex < N; PriceIndex++)
  {
    Price[PriceIndex] = pu_f[i][k] *P_old[PriceIndex][k + fv_up]+pd_f[
  }

/*Set Rhs*/
S[1] = B1[1] * Price[1] + C1[1] * Price[2] + A1[1] * bound1 - alpha *
for (PriceIndex = 2; PriceIndex < N - 1; PriceIndex++)
  S[PriceIndex] = A1[PriceIndex] * Price[PriceIndex - 1] +
                  B1[PriceIndex] * Price[PriceIndex] +
                  C1[PriceIndex] * Price[PriceIndex + 1];
S[N - 1] = A1[N - 1] * Price[N - 2] + B1[N - 1] * Price[N - 1] + C1[N

//Add M(t)S
for (PriceIndex = 1; PriceIndex < N; PriceIndex++)
```

```
  {
    S[PriceIndex] += compute_S(vect_y[PriceIndex], V[i][k], omega, rho
  }

/*Solve the system*/
for (PriceIndex = N - 2; PriceIndex >= 1; PriceIndex--)
  S[PriceIndex] = S[PriceIndex] - C[PriceIndex] * S[PriceIndex + 1];

Price[1] = S[1] / B[1];

for (PriceIndex = 2; PriceIndex < N; PriceIndex++)
  Price[PriceIndex] = S[PriceIndex] / B[PriceIndex] - A[PriceIndex] *

for (PriceIndex = 1; PriceIndex < N; PriceIndex++)
  {
    P_new[PriceIndex][k] = discount * Price[PriceIndex];
  }

//Monitoring
if (flag_monit)
  {
    for (PriceIndex = 1; PriceIndex < N; PriceIndex++)
      Price[PriceIndex] = P_new[PriceIndex][k];

    if (ratchet)
      {
        for (PriceIndex = 1; PriceIndex < N; PriceIndex++)
          {
            val_s = compute_S(vect_y[PriceIndex], V[i][k], omega, rho)
            new_A = MAX(val_s, A_IN);
            s_value = A_IN / new_A * val_s;
            i_s = 0;
            i_s = 0;
            while ((compute_S(vect_y[i_s], V[i][k], omega, rho) < s_va

            if (i_s == 0)
              new_P = Price[1];
            else if (i_s >= N)
              new_P = Price[N - 1];
            else
              {
```

```
                    val = Price[i_s];
                    val1 = Price[i_s - 1];
                    val_s = compute_S(vect_y[i_s], V[i][k], omega, rho);
                    val_sim = compute_S(vect_y[i_s - 1], V[i][k], omega, r
                    new_P = val + (val - val1) * (s_value - val_s) / (val_
                  }

                P_new[PriceIndex][k] = new_A / A_IN * new_P;
Price[PriceIndex]=P_new[PriceIndex][k];
              }
          }

        if (fabs(gamma_i) < 0.000001) //Bonus Event
          {
            for (PriceIndex = 1; PriceIndex < N; PriceIndex++)
              {
                new_A = A_IN * (1 + Bt[anno]);
                val_s = compute_S(vect_y[PriceIndex], V[i][k], omega, rho)
                s_value = A_IN / new_A * val_s;
                if (i_s == 0)
                  new_P = Price[1];
                else if (i_s >= N)
                  new_P = Price[N - 1];
                else
                  {
                    val = Price[i_s];
                    val1 = Price[i_s - 1];
                    val_s = compute_S(vect_y[i_s], V[i][k], omega, rho);
                    val_sim = compute_S(vect_y[i_s - 1], V[i][k], omega, r
                    new_P = val + (val - val1) * (s_value - val_s) / (val_
                  }
              }
            P_new[PriceIndex][k] = MAX(P_new[PriceIndex][k], new_A / A_IN
          }
        else if (gamma_i <= 1) //Withdrawal not exceeding contract amount
          {
            for (PriceIndex = 1; PriceIndex < N; PriceIndex++)
              {
                val_s = compute_S(vect_y[PriceIndex], V[i][k], omega, rho)
                s_value = MAX(val_s - gamma_i * Gt[anno] * A_IN, 0.);
                i_s = 0;
```

```
        while ((compute_S(vect_y[i_s], V[i][k], omega, rho) < s_va

        if (i_s == 0)
          new_P = Price[1];
        else if (i_s >= N)
          new_P = Price[N - 1];
        else
          {
            val = Price[i_s];
            val1 = Price[i_s - 1];
            val_s = compute_S(vect_y[i_s], V[i][k], omega, rho);
            val_sim = compute_S(vect_y[i_s - 1], V[i][k], omega, r
            new_P = val + (val - val1) * (s_value - val_s) / (val_
          }
        P_new[PriceIndex][k] = new_P + Rt[anno] * gamma_i * Gt[ann
      }
  }
else if (gamma_i > 1.) //Partial or full surrender
  //gamma_i=2 is full surrender
  {
    for (PriceIndex = 1; PriceIndex < N; PriceIndex++)
      {
        new_A = A_IN * (2 - gamma_i);
        val_s = compute_S(vect_y[PriceIndex], V[i][k], omega, rho)
        s_value1 = MAX(val_s - Gt[anno] * A_IN, 0.);
        s_value = s_value1 * (2 - gamma_i);
        if (i_s == 0)
          new_P = Price[1];
        else if (i_s >= N)
          new_P = Price[N - 1];
        else
          {
            val = Price[i_s];
            val1 = Price[i_s - 1];
            val_s = compute_S(vect_y[i_s], V[i][k], omega, rho);
            val_sim = compute_S(vect_y[i_s - 1], V[i][k], omega, r
            new_P = val + (val - val1) * (s_value - val_s) / (val_
          }
        P_new[PriceIndex][k] = MAX(P_new[PriceIndex][k], new_A / A
      }//end PriceIndex
  }
```

```
            }//End Monitoring
          }//end k

        //Copy
        for (j = 0; j <= N; j++)
          for (k = 0; k <= i; k++)
            P_old[j][k] = P_new[j][k];

      }//end i

    Index = (int) floor((double)N / 2.0);
    price = P_new[Index][0] - s0;

    //Memory Desallocation
    free(A);
    free(B);
    free(C);
    free(A1);
    free(B1);
    free(C1);
    free(vect_y);
    free(S);
    free(Price);

    return price;

}

/*Compute Price Option*/
int  Fd_HybridTree_GLWB_Hes(double s, double t_fixed, double r_fixed, double div
{

    int i,j,iter;
    int dummy;
    double val, sum, sum1;
    double pr1, pr2, pracc;
    FILE *f_in;

    n_step_for_period = n_step_for_year;
    Nt = (int)(t_fixed * n_step_for_year);
    N = N_space;
```

```
theta_sc = theta_fd;

tt = t_fixed;
s0 = s;
divid = divid_fixed;
r = r_fixed;
alpha_m = alpha_m_fixed;
ratchet = ratchet_fixed;
gamma_i = gamma_i_fixed;
theta = theta_fixed;
kappa = kappa_fixed;
omega = omega_fixed;
rho = rho_fixed;
v0 = v0_fixed;

f_in = fopen(infilename, "rb");

for (i = 0; i <= 60; i++)
  {
    fscanf(f_in, "%lf\ n", &val);
    lm_insurance[i] = val;
    lm_age[i] = 65. + i;
  }

sum = 1.;
Mt[0] = lm_insurance[0];
for (i = 1; i <= 60; i++)
  {
    sum *= (1. - lm_insurance[i - 1]);
    Mt[i] = lm_insurance[i] * sum;
  }

Rt[0] = 1.;
sum1 = 0;
for (i = 1; i <= 60; i++)
  {
    sum1 += Mt[i - 1];
    Rt[i] = 1. - sum1;
  }

//Surrender fee Kt
```

```
   for (i = 0; i <= 60; i++)
    {
  j=0;
  while((pnl_vect_get(timesKt_fixed,j)<(double)i)&&(j<=4))
j++;
  Kt[i] = pnl_vect_get(Kt_fixed,j);
    }

  for (i = 1; i <= 60; i++)
    Bt[i] = B_i_fixed;

  for (i = 1; i <= 60; i++)
    Gt[i] = Gt_i_fixed;


  /*Memory Allocation*/
  dummy = memory_allocation(Nt, N);
  if (dummy != OK) return FAIL;

  //Tree construction for v
  tree_v(tt, v0, kappa, theta, omega, Nt);

  iter = 1;

  if (iter)
    {
      pracc = 1.e-8;
      pr1 = 0.;
      pr2 = 0.1;
      val = rtsec(compute_price, pr1, pr2, pracc);
    }

  /*Price*/
  *ptprice = val;

  fclose(f_in);

  /*Memory Disallocation*/
  free_memory(Nt, N);

  return OK;
```

```
}


int CALC(FD_HybridTree_GLWB_HES)(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.);
  infilename = ptOpt->MortalityData.Val.V_FILENAME;

  return Fd_HybridTree_GLWB_Hes(ptMod->S0.Val.V_PDOUBLE,
                                ptOpt->Maturity.Val.V_DATE - ptMod->T.Val.V_DATE
                                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_PINT,
                                Met->Par[1].Val.V_PINT,
                                Met->Par[2].Val.V_RGDOUBLE051,
                                &(Met->Res[0].Val.V_DOUBLE));
}

static int CHK_OPT(FD_HybridTree_GLWB_HES)(void *Opt, void *Mod)
{
  if ((strcmp(((Option *)Opt)->Name, "GLWB") == 0))
    return OK;
  else
    return WRONG;
}
#endif //PremiaCurrentVersion

static int MET(Init)(PricingMethod *Met, Option *Opt)
{
  if (Met->init == 0)
    {
      Met->init = 1;
      Met->HelpFilenameHint = "fd_hybridtree_glwb_hes";
```

```
      Met->Par[0].Val.V_INT = 10;
      Met->Par[1].Val.V_INT = 200;
      Met->Par[2].Val.V_RGDOUBLE = 0.5;
    }

  return OK;
}

PricingMethod MET(FD_HybridTree_GLWB_HES) =
{
  "FD_HybridTree_GLWB_HES",
  {
    {"Timestep_for_year", INT2, {100}, ALLOW}, {"SpaceStepNumber", INT2, {100},
    {"Theta", RGDOUBLE051, {100}, ALLOW},
    {" ", PREMIA_NULLTYPE, {0}, FORBID}
  },
  CALC(FD_HybridTree_GLWB_HES),
  { {"Fair Fee", DOUBLE, {100}, FORBID},
    {" ", PREMIA_NULLTYPE, {0}, FORBID}
  },
  CHK_OPT(FD_HybridTree_GLWB_HES),
  CHK_ok,
  MET(Init)
};
```