Help

```
#include  "bharchiarella1d_stdi.h"
#include "enums.h"

#if defined(PremiaCurrentVersion) && PremiaCurrentVersion <
     (2009+2) //The "#else" part of the code will be freely av
    ailable after the (year of creation of this file + 2)
static int CHK_OPT(MC_BC_TEICHMANNBAYER)(void *Opt, void *
    Mod)
{
  return NONACTIVE;
}
int CALC(MC_BC_TEICHMANNBAYER)(void*Opt,void *Mod,Pricing
    Method *Met)
{
  return AVAILABLE_IN_FULL_PREMIA;
}
#else

/* linear uniform interpolation of [0,T] of size N*/
/* return value = dt*/
static double linspace1(double T0, double T1, int N,
    double* t )
{
  double dt;
  int i;

  dt=(T1 - T0) / (double)( N - 1 );
  t[0] = T0;
  for (i=1; i<N; i++)
    t[i] = t[i-1] + dt;
  return dt;
}
/* linear interpolation using stepsize dt; return T */
static double linspace2( double dt, int N, double* t ){
  double T = dt * (double)( N-1 );
  int i;
  t[0] = 0.0;
  for (i=1; i<N; i++)
    t[i] = t[i-1] + dt;
```

```
  return T;
}


static void BhCh( double b_0, double b_1, double eta, cons
    t double* x, int l, double* r ){
  int i;
  for( i=0; i<l; i++ )
    r[i] = b_0 + b_1 * (1.0 - exp(-eta*x[i]));
}

/* Generate an Nxd-array of iid. Bernoullis with p = 0.5 */
/* actually, it would be enough to consider boolean variab
    les here */
static void GenBernoulli1( int* J, int N,int generator)
{
  int i;
  for( i=0; i<N; i++)
    if (pnl_rand_uni(generator)< 0.5)
      J[i] = 0;
    else  J[i] = 1;

}

static void CopyVect( const double* orig, double* dest,
    int N ){
  int i;
  for (i=0; i<N; i++ )
    dest[i] = orig[i];
}

/* generate a vector of "brownian increments" given the mu
    lti-index J */
static void omegadot1( int N, double dt, int NCub, const
    int* J, int n, double* dB ){
  int i,k;
  double tempd1 = sqrt(dt) / sqrt((double)(n));
  for(i=0; i<(NCub-1); i++ ){
    for (k = 0; k<n; k++ )
      dB[i*n+k] = (J[i] == 0) ? tempd1 : (- tempd1);
  }
```

```
  for ( k = (n*(NCub-1)); k<N; k++ )
    dB[k] = (J[NCub-1] == 0) ? tempd1 : (-tempd1);
}

static double Shift( const double* r, const double* x,
    double dx, double dt, int m, int k, int i_shift, double r_shift
    ){
  /*double ret;*/
  if (k < m - i_shift - 1 )
    return (1.0-r_shift) * r[k+i_shift] + r_shift * r[k+i_
    shift+1];
  else
    return r[m-1];
}

static double alpha0( const double* r, double a_0, double
    a_r, double a_f, double gamma, double lambda, double t,
    double tau, int tau_shift, const double* x, int m, int k,
    double expl){
  double temp = MAX(0.,a_0 + a_r * r[0] + a_f * r[tau_shif
    t]);
  return gamma * pow(temp,2.0*gamma) * (1.0 - expl) * expl
    / lambda - 0.5 * ( gamma * pow(temp,gamma-1.0) * ( a_r *
    pow(temp,gamma) + a_f * pow(temp,gamma) * exp( - lambda * (
    2.0 * t + x[k] - tau) ) ) * expl);
}


static double HJMSigma( const double* r, double a_0,
    double a_r, double a_f, double gamma0, double lambda, int tau_sh
    ift, const double* x, int m, int k, double expl ){
  return pow( MAX(a_0 + a_r * r[0] + a_f * r[tau_shift],0.)
    , gamma0 ) * expl;
}


/* value P(0,T) of a zero coupon bond */
static double ZeroCB( const double* r, const double* x,
    double dx, double T ){
  int Tx = ceil( T / dx ); /* index of T in the x-grid */
  double integ = 0.0;
```

```
  int i;
  for (i=0; i<Tx; i++ )
    integ += 0.5 * ( r[i] + r[i+1] ) * dx;
  return exp( - integ );
}

/* compute the empirical mean value of a vector */
static double mean( const double* X, int M ){
  double ret = 0.0;
  int i;
  for (i=0; i<M; i++ )
    ret += X[i];
  ret = ret / (double)(M);
  return ret;
}

/* compute the empirical standard deviation of a vector */
static double stdev( const double* X, int M ){
  double mu = mean( X, M );
  double ret = 0.0;
  int i;
  for(i=0; i<M; i++ )
    ret += X[i]*X[i];
  ret = ret / (double)(M);
  ret = sqrt( ret - mu * mu );
  return ret;
}


/* n number of time intervals on each cubature interval*/
/* N number of time intervals*/
/* m number of space intervals*/
/* M number of paths for Monte-Carlo simulation*/
static int mc_bc_teichmannbayer(double a_0,double a_r,
    double a_f, double gamma0, double lambda,double b_0,double b_1,
    double eta,double tau,double t0,  double T_bond, double T_
    option,NumFunc_1 *p,int generator,int n,int L,int k, int M,
    double *price,double *error)
{

  int NCub;
```

```
double *t, dt;
double *x, dx;
int mAct;
double r_shift;
int i_shift;
int *J;
double *r;
double *rp,*rm,*rpc,*rmc;
double *res,*dB;
double Bp, Bm; /* savings account */
double expl; /* auxiliary variables */
double zerop;
double zerom;
int j, i, k_bis;
int tau_shift;

/*tau appears in forward rate volatility description*/
if(tau>T_bond)
  return PREMIA_UNTREATED_TAU_BHAR_CHIARELLA;

pnl_rand_init(generator,1,M);
//K=p->Par[0].Val.V_DOUBLE;



/* first generate the time and space grids */
if (L % n == 0)
  NCub = L / n;
else
  NCub = L / n + 1;
/* generate the time grid */
t=malloc((L+1)*sizeof(double));
dt = linspace1( t0, T_option, L+1, t );
/* gegerate the spacegrid */
dx = ( T_bond - T_option ) / (double)( k );
mAct = (int)( T_option / dx ) + k + (int)(L * (dt/dx)) +
  1;
x=malloc((mAct+1)*sizeof(double));
dt = linspace1( t0, T_option, L+1, t );
linspace2( dx, mAct + 1, x );
/* for the shift semigroup, express dt in temrs of dx:
```

```
 dt = i_shift * dx + r_shift * dx */
r_shift = dt / dx;
i_shift = (int)( r_shift );
r_shift = r_shift - i_shift;

/* J describes one cubature path */
J=malloc((NCub)*sizeof(int));

/* generate the initial forward rate curve */
r=malloc((mAct+1)*sizeof(double));/* saves initial
                                   * forward rate
                                   curve */


BhCh( b_0, b_1, eta, x, mAct+1, r );

rp=malloc((mAct+1)*sizeof(double));
rm=malloc((mAct+1)*sizeof(double));
rpc=malloc((mAct+1)*sizeof(double));
rmc=malloc((mAct+1)*sizeof(double));

/* the path-wise discounted payoff */
res =malloc((M)*sizeof(double));

/* the "brownian" increments (i.e. the cubature derivati
  ves) */
dB =malloc((L)*sizeof(double));

/* now iterate through all paths for the MC-simulation*/
for(j=0; j<M; j++ ){
  /* re-initialize r and B */
  GenBernoulli1( J, NCub,generator ); /* generate J */
  CopyVect( r, rp, mAct+1 );
  CopyVect( r, rm, mAct+1 );
  Bp = 1.0;
  Bm = 1.0;

  /* generate dB */
  omegadot1( L, dt, NCub, J, n, dB );

  /* iterate through the time grid */
```

```
  for(i=0; i<L; i++ )
    {
      tau_shift = (int)((tau - t[i]) * (dt / dx));
      Bp += Bp * rp[0] * dt;
      Bm += Bm * rm[0] * dt;
      CopyVect( rp, rpc, mAct+1 );
      CopyVect( rm, rmc, mAct+1 );
      /* iterate through the space grid */
      for (k_bis=0; k_bis<=mAct; k_bis++){
        expl = exp( - lambda * x[k_bis] );

        rp[k_bis] = Shift( rpc, x, dx, dt, mAct+1, k_bis,
   i_shift, r_shift ) + alpha0( rpc, a_0, a_r, a_f, gamma0,
  lambda, t[i], tau, tau_shift, x, mAct+1, k_bis, expl ) * dt;

        rp[k_bis] += HJMSigma( rpc, a_0, a_r, a_f, gamma0
  , lambda, tau_shift, x, mAct+1, k_bis, expl ) * dB[i];
        rm[k_bis] = Shift( rmc, x, dx, dt, mAct+1, k_bis,
   i_shift, r_shift ) + alpha0( rmc, a_0, a_r, a_f, gamma0,
  lambda, t[i], tau, tau_shift, x, mAct+1, k_bis, expl ) * dt;

        rm[k_bis] -= HJMSigma( rmc, a_0, a_r, a_f, gamma0
  , lambda, tau_shift, x, mAct+1, k_bis, expl ) * dB[i];
      }
    }


  /* compute the discounted payoff for this particular
  path */
  zerop = ZeroCB(rp, x, dx, T_bond - T_option );
  zerom = ZeroCB(rm, x, dx, T_bond - T_option );

  res[j]=0.5*((p->Compute)(p->Par,zerop)/Bp+(p->Compute)(
  p->Par,zerom)/ Bm);
}

*price = mean( res, M );
*error = 1.65 * stdev( res, M ) / sqrt( (double)(M) );

/* free memory again */
free(t);
```

```
  free(x);
  free(J);
  free(r);
  free(rp);
  free(rpc);
  free(rm);
  free(rmc);
  free(res);
  free(dB);

  return OK;

}

int CALC(MC_BC_TEICHMANNBAYER)(void *Opt,void *Mod,Pricing
    Method *Met)
{
  TYPEOPT* ptOpt=(TYPEOPT*)Opt;
  TYPEMOD* ptMod=(TYPEMOD*)Mod;

  return mc_bc_teichmannbayer(ptMod->alpha0.Val.V_PDOUBLE,
    ptMod->alphar.Val.V_PDOUBLE,ptMod->alphaf.Val.V_PDOUBLE,pt
    Mod->gamm.Val.V_PDOUBLE,ptMod->lambda.Val.V_PDOUBLE,ptMod->
    beta0.Val.V_PDOUBLE,ptMod->beta1.Val.V_PDOUBLE,ptMod->eta.
    Val.V_PDOUBLE,ptMod->tau.Val.V_PDOUBLE,ptMod->T.Val.V_DATE,
    ptOpt->BMaturity.Val.V_DATE,ptOpt->OMaturity.Val.V_DATE,pt
    Opt->PayOff.Val.V_NUMFUNC_1, Met->Par[0].Val.V_ENUM.value,
    Met->Par[1].Val.V_PINT,Met->Par[2].Val.V_PINT,Met->Par[3].Val
    .V_PINT,Met->Par[4].Val.V_PINT,&(Met->Res[0].Val.V_DOUBLE)
    ,&(Met->Res[1].Val.V_DOUBLE));
}

static int CHK_OPT(MC_BC_TEICHMANNBAYER)(void *Opt, void *
    Mod)
{
  if ((strcmp(((Option*)Opt)->Name,"ZeroCouponCallBondEuro"
    )==0)|| (strcmp(((Option*)Opt)->Name,"ZeroCouponPutBondEu
    ro")==0))
    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_ENUM.value=0;
      Met->Par[0].Val.V_ENUM.members=&PremiaEnumMCRNGs;
      Met->Par[1].Val.V_PINT=20;
      Met->Par[2].Val.V_PINT=400;
      Met->Par[3].Val.V_PINT=10;
      Met->Par[4].Val.V_PINT=20;
    }

  return OK;
}

PricingMethod MET(MC_BC_TEICHMANNBAYER)=
{
  "MC_BC_TEICHMANNBAYER",

  {{"RandomGenerator",ENUM,{100},ALLOW},
   {"Number of time intervals on each cubature interval",
    INT,{100},ALLOW},
   {"Number of time intervals",INT,{100},ALLOW},
   {"Number of space intervals*",INT,{100},ALLOW},
   {"Number of paths for Monte-Carlo simulation",PINT,{100}
    ,ALLOW},
   {" ",PREMIA_NULLTYPE,{0},FORBID}},
  CALC(MC_BC_TEICHMANNBAYER),
  {{"Price",DOUBLE,{100},FORBID},{"MC Error",DOUBLE,{100},
    FORBID} ,{" ",PREMIA_NULLTYPE,{0},FORBID}},
  CHK_OPT(MC_BC_TEICHMANNBAYER),
  CHK_ok,
  MET(Init)
} ;
```

References