Help

```
#include  "cir1d_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_TEICHMANNBAYER)(void *Opt, void *Mod)
{
  return NONACTIVE;
}
int CALC(MC_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;
```

```
}

/* extrapolate a CIR-HJM-forward rate curve from a short ra
    te */
/* l is assumed to be the length of x */
static void CIR2HJM( double r0, double kappa, double theta,
     double sigma, double gamma_tb, const double* x, int l,
    double* r ){
  double g0, g1, G1;
  int i;
  for (i=0; i<l; i++ ) {
    G1 = 2.0 * (exp( gamma_tb * x[i] ) - 1.0) / (( gamma_tb
     + kappa ) * (exp( gamma_tb * x[i] ) - 1.0) + 2.0 * gamma_
    tb );
    g0 = kappa * theta * G1;
    g1 = 4.0 * (gamma_tb * gamma_tb) * exp( gamma_tb * x[i]
      ) / (( ( gamma_tb + kappa ) * exp( gamma_tb * x[i] ) +
    gamma_tb - kappa ) * ( ( gamma_tb + kappa ) * exp( gamma_tb
    * x[i] ) + gamma_tb - kappa ));
    r[i] = g0 + r0 * g1;
  }
}

/* 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;

}

/* 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);
}

/* apply the shift semi-group */
/* i_shift is the integer part of the shift in terms of dx-
    units, r_shift
 the remainder */
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];
}

/* the vector field alpha0 */
static double alpha0( const double* r, double kappa,
    double theta, double sigma, double gamma_tb, const double* x,
    int m, int k, double expg){
  double g1, G1;
  G1 = 2.0 * (expg - 1.0) / (( gamma_tb + kappa ) * (expg -
     1.0) + 2.0 * gamma_tb );
  g1 = 4.0 * (gamma_tb * gamma_tb) * expg / (( ( gamma_tb +
     kappa ) * expg + gamma_tb - kappa ) * ( ( gamma_tb + kapp
    a ) * expg + gamma_tb - kappa ));
  return (sigma * sigma) * (g1) * ( r[0] * (G1) - 0.25 );
}

/* the volatility vector field(s) */
double HJMSigma( const double* r, double kappa, double thet
    a, double sigma, double gamma, const double* x, int m, int
    k, double expg, double sqrtr ){
```

```
  return sigma * sqrtr * ( 4.0 * (gamma * gamma) * expg / (
    ( ( gamma + kappa ) * expg + gamma - kappa ) * ( ( gamma +
     kappa ) * expg + gamma - kappa )));
}

/* 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_teichmannbayer(double r0, double kappa,
    double t0, double sigma,double theta, double T_bond, double T_
    option,NumFunc_1 *p,int generator,int n,int L,int k, int M,
    double *price,double *error)
{
  double gamma_tb = 0;
  int NCub;
  double *t;
  double dt = 0;
  double dx = 0;
  int mAct = 0;
  double *x;
  double r_shift = 0;
  int i_shift = 0;
  int *J;
  double *r;
  double *rp,*rm,*rpc,*rmc;
  double *res,*dB;
  double Bp, Bm; /* savings account */
  double expg, sqrtrm, sqrtrp; /* auxiliary variable*/
  int i,j;
  int ii,k_bis;
  double zerop = 0;
  double zerom = 0;

  gamma_tb = sqrt( (kappa*kappa) + 2.0 * (sigma * sigma) );
  pnl_rand_init(generator,1,M);

/* 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 );
      /* generate 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 fo
rward rate curve */
  CIR2HJM( r0, kappa, theta, sigma, gamma_tb, 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 deriv
atives) */
  dB =malloc((L)*sizeof(double));

  /* now iterate through all paths for the MC-simulatio
n*/
  for(j=0; j<M; j++ ){
    /* re-initialize r and B */
    GenBernoulli1( J, NCub,generator ); /* generate J *
/

    for (i=0; i<mAct+1; i++ )
      {
```

```
          rp[i]=r[i];
          rm[i]=r[i];
        }

    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++ ){
      sqrtrp = (rp[0] > 0.0) ? sqrt(rp[0]) : 0.0;
      sqrtrm = (rm[0] > 0.0) ? sqrt(rm[0]) : 0.0;
      Bp += Bp * rp[0] * dt;
      Bm += Bm * rm[0] * dt;
      for (ii=0; ii<mAct+1; ii++ )
        {
          rpc[ii]=rp[ii];
          rmc[ii]=rm[ii];
        }

      /* iterate through the space grid */
      for (k_bis=0; k_bis<=mAct; k_bis++){
        expg = exp( gamma_tb * x[k_bis] );
        rp[k_bis] = Shift( rpc, x, dx, dt, mAct+1, k_bi
s, i_shift, r_shift ) + alpha0( rpc, kappa, theta, sigma,
gamma_tb, x, mAct+1, k_bis, expg ) * dt;
        rp[k_bis] += HJMSigma( rpc, kappa, theta, sigma
, gamma_tb, x, mAct+1, k_bis, expg, sqrtrp ) * dB[i];
        rm[k_bis] = Shift( rmc, x, dx, dt, mAct+1, k_bi
s, i_shift, r_shift ) + alpha0( rmc, kappa, theta, sigma,
gamma_tb, x, mAct+1, k_bis, expg ) * dt;
        rm[k_bis] -= HJMSigma( rmc, kappa, theta, sigma
, gamma_tb, x, mAct+1, k_bis, expg,sqrtrm) * dB[i];
      }
    }
    /* compute the discounted payoff for this particul
ar 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->Compu
    te)(p->Par,zerom)/ Bm);
      }
      /*Price*/
      *price=mean( res, M );

      /*Estimate Error*/
      *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_TEICHMANNBAYER)(void *Opt,void *Mod,Pricing
    Method *Met)
{
  TYPEOPT* ptOpt=(TYPEOPT*)Opt;
  TYPEMOD* ptMod=(TYPEMOD*)Mod;

  return mc_teichmannbayer(ptMod->r0.Val.V_PDOUBLE,ptMod->
    k.Val.V_DOUBLE,ptMod->T.Val.V_DATE,ptMod->Sigma.Val.V_PDOUB
    LE,
                           ptMod->theta.Val.V_PDOUBLE,pt
    Opt->BMaturity.Val.V_DATE,ptOpt->OMaturity.Val.V_DATE,ptOpt->
    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_PI
    NT,Met->Par[4].Val.V_PINT,&(Met->Res[0].Val.V_DOUBLE),&(
    Met->Res[1].Val.V_DOUBLE));
}
```

```
static int CHK_OPT(MC_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_TEICHMANNBAYER)=
{
  "MC_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_TEICHMANNBAYER),
```

```
  {{"Price",DOUBLE,{100},FORBID},{"MC Error",DOUBLE,{100},
    FORBID} ,{" ",PREMIA_NULLTYPE,{0},FORBID}},
  CHK_OPT(MC_TEICHMANNBAYER),
  CHK_ok,
  MET(Init)
} ;
```

References