Help

```
extern "C" {
#include  "
href../../../mod/guyon1d/guyon1d_std/guyon1d_std_h_src.pdfguyon1d_std.h"
#include "pnl/pnl_matrix.h"
#include "pnl/pnl_mathtools.h"
#include "pnl/pnl_vector.h"
#include "pnl/pnl_random.h"
#include "pnl/pnl_finance.h"
#include "
href../../../common/enums_h_src.pdfenums.h"

#if defined(PremiaCurrentVersion) && PremiaCurrentVersion < (2017+2) //The "#els
static int CHK_OPT(MC_Guyon)(void *Opt, void *Mod)
{
  return NONACTIVE;
}

int CALC(MC_Guyon)(void *Opt, void *Mod, PricingMethod *Met)
{
  return AVAILABLE_IN_FULL_PREMIA;
}
#else

static void  Algo1(PnlVect *Ei, PnlVect *Si, PnlVect *Vi, int L, int j){ // Comp
    double S1,S2;
    int M = Si->size; // number of MC samples
    int size_bin = floor((double)(M-1) / L); // size of a bin

    PnlVectInt *index = pnl_vect_int_create(M);

    pnl_vect_qsort_index(Si, index, 'i'); // sort Si
    pnl_vect_permute_inplace(Vi, index); // sort Vi like Si

    for (int l = 0; l < L; l++) // for each bin
  {
        S1 = 0;
        S2 = 0;
        for (int k = 1; k < size_bin + 1; k++) // for each S_i,j in the l-th bin
  {
```

```
            S1 += pnl_vect_get(Vi, l * size_bin + k)*pnl_vect_get(Vi, l * size_b
            S2 += 1.0;
  }
        pnl_vect_set(Ei, l, (double) S1 / S2);
  }
    pnl_vect_int_free(&index);
}


  static double computeSigmaPdv(int i,int j, double sigma_u,double sigma_l, PnlM
  //p4 of the article "path dependent volatility"
  {

    double X;

X = pnl_mat_get(S, i, j-MIN(j,N_step));
  if (pnl_mat_get(S, i, j) <= X )   {
return sigma_u;
  }
  else{
return sigma_l;
  }

    // if(example == 1 || example == 4){
//   X = pnl_mat_get(S, i, j-MIN(j,N_step));
    // }

    // else if(example == 2 || example == 5){
//   X = 0.0;
//   for (int k=0;k<MIN(N_step,j)+1;k++ )
//  X += pnl_mat_get(S, i, j-k);
//   X *= (double)(N_step+1)/N_step/12;
    // }

    // else if(example == 3 || example == 6){

//   m = pnl_mat_get(S, i, j);
//   M = pnl_mat_get(S, i, j);
//   for (int k=1;k<MIN(N_step,j)+1;k++){

//  m = MIN(m,pnl_mat_get(S, i, j-k));
```

```
//  M = MAX(M,pnl_mat_get(S, i, j-k));
//   }
    // }

    // if (example == 1 || example == 2){

//   if (pnl_mat_get(S, i, j) <= X )   {
//  return sigma_u;
//   }
//   else{
//  return sigma_l;
//   }
    // }
    // else if (example == 3){

//   if ((pnl_mat_get(S, i, j)-m)/(M-m) <= 0.5)
//  return sigma_u;
//   else
//  return sigma_l;
    // }
    // else if (example == 4 || example == 5){
//   if(ABS(pnl_mat_get(S, i, j)/X -1.0)> 0.02/sqrt((double)12*N_step))
//  return sigma_u;
//   else
//  return sigma_l;
    // }
    // else if (example == 6){
//   if(M/m-1>0.5*sigma_u*sqrt((double)MIN(j,N_step)/N_step/12))
//  return sigma_u;
//   else
//  return sigma_l;
    // }
  }

  // Returns sigma_ATM p6 of the article "path dependent volatility"
  static double compute_sigma_ATM(double eps,double esp,double s,double K,double
  {
double sigma_min = 0.0;
double sigma_max = 100.0;
while(pnl_bs_call (s,K,T, r,divid,sigma_max)<esp){
  sigma_max *=2.0;
```

```
    }
    while (sigma_max-sigma_min>eps){
  if (pnl_bs_call (s,K,T, r,divid,(sigma_max+sigma_min)/2.0)>esp){
sigma_max = (sigma_max+sigma_min)/2.0;
  }
  else{
sigma_min = (sigma_max+sigma_min)/2.0;
  }
    }
    return (sigma_min+sigma_max)/2.0;
  }

  // Return price = delta_sigma of the forward call spreads , p6 of the article
  static double compute_delta_sigma(double eps,double esp_call_spread,double s,d
  {
    double delta_min = -2.0*sigma_atm;
    double delta_max = 2.0*sigma_atm;

    while (delta_max-delta_min>eps){
  if ((pnl_bs_call (s,K1,T, r,divid,sigma_atm+((delta_max+delta_min)/4.0))-pnl_b
delta_max = (delta_min+delta_max)/2.0;
  }
  else{
delta_min = (delta_min+delta_max)/2.0;
  }
    }
    return (delta_min+delta_max)/2.0;
  }

  int MCGuyon(double S0, NumFunc_1  *p, double TT,double sigma_l, double sigma_u
  {
    double T = nmonit*TT;  // maturity (months)
    int N_step = 10; // number of time steps per month
double K1 = p->Par[0].Val.V_DOUBLE;
double K2 = p->Par[1].Val.V_DOUBLE;
    double eps = 1.0E-10; // precision used to compute local volatility
    int example = 1; // which path dependent volatility do we use
    int M = L*multipl + 1; // number of Monte Carlo samples
    int N = N_step*T; // number of time steps

    PnlVect *sigma_ATM = pnl_vect_create_from_scalar (N+1, 0.0); // ATM implied
```

```
PnlVect *delta_sigma = pnl_vect_create_from_scalar (N+1, 0.0); // delta_sigm

PnlMat *Z1, *S, *V; // Z1 sample from the normal law, S matrix of the asset
PnlRng *rng = pnl_rng_create(PNL_RNG_KNUTH);

// Initialization

Z1 = pnl_mat_new();
V = pnl_mat_new();
S = pnl_mat_new();

//pnl_rng_sseed(rng, 15646);

pnl_mat_resize(Z1, M, N);
pnl_mat_set_zero(Z1);

pnl_mat_resize(V, M, N + 1);
pnl_mat_set_all(V, sigma_u);

pnl_mat_resize(S, M, N + 1);
pnl_mat_set_all(S, S0);

pnl_mat_rng_normal(Z1, M, N, rng);

PnlVect *Ej = pnl_vect_create_from_double(L, 0.0); // Ei : conditional expec
PnlVect *Vj = pnl_vect_create_from_double(M, 0.0); // Vi : pdv volatility fo
PnlVect *Sj = pnl_vect_create_from_double(M, 0.0); // Si : asset price for e

double vij, sij, EC;
int i, j;

for (j = 0; j < N; j++) // for each time step
{

    pnl_mat_get_col(Vj, V, j);
    pnl_mat_get_col(Sj, S, j);
    Algo1(Ej, Sj, Vj, L, j); // sort Sj and Vj, gives conditional expectatio

    for (i = 0; i < M; i++) // for each Monte Carlo Sample
    {
        vij = pnl_mat_get(V, i, j);
```

```
            sij = pnl_mat_get(S, i, j);

            // take conditional expectation from the right bin
            if (i == 0 ) EC = pnl_vect_get(Ej,0);
            else EC = pnl_vect_get(Ej, floor((double)(i - 1) * L / (M - 1)));

            // Compute S_i,j+1
            pnl_mat_set(S, i, j + 1, sij*exp( sigma_dup*vij/sqrt(EC)/sqrt((doubl

            // Compute Sigma i,j+1
            pnl_mat_set(V, i, j + 1, computeSigmaPdv(i,j+1,sigma_u,sigma_l,S,N_s

        }
    }

    double esp = 0.0; // (S_t,i/S_t,i-1 -1)+ expectation
    double var = 0.0; // (S_t,i/S_t,i-1 -1)+ variance
    double esp_call_spread = 0.0; // (S_t,i/S_t,i-1 -K1)+ - (S_t,i/S_t,i-1 -K2)+
    double var_call_spread = 0.0; // (S_t,i/S_t,i-1 -K1)+ - (S_t,i/S_t,i-1 -K2)+
    // Computing implied volatility to get prices
    double K = 1; // strike
    double r = 0.0; // interest rate
    double divid = 0.0; // dividend rate
double delta_sig;

    for (int j =0;j<(int)T;j++){ // for each time step
   esp = 0.0;
        var = 0.0;
        esp_call_spread = 0.0;
        var_call_spread = 0.0;
        for (int i=0;i<M;i++){ // for each MC sample

            esp += MAX((pnl_mat_get(S, i, (j+1)*N_step)/pnl_mat_get(S, i, j*N_st
            var += MAX((pnl_mat_get(S, i, (j+1)*N_step)/pnl_mat_get(S, i, j*N_st

            esp_call_spread += MAX((pnl_mat_get(S, i, (j+1)*N_step)/pnl_mat_get(

            double var_j = MAX((pnl_mat_get(S, i, (j+1)*N_step)/pnl_mat_get(S, i
            var_call_spread += var_j*var_j;
        }
```

```
        esp /= (double)M;
        esp_call_spread /= (double)M;

        var /= (double)M;
        var_call_spread /= (double)M;
        var -= esp*esp;
        var_call_spread -= esp_call_spread*esp_call_spread;

        pnl_vect_set (sigma_ATM,j+1,compute_sigma_ATM(eps,esp,1,1,1./(double)nmo

        delta_sig = compute_delta_sigma(eps,esp_call_spread,1,K1/S0,K2/S0,1./(do

        pnl_vect_set (delta_sigma,j+1,delta_sig);

    }

*ptprice=esp_call_spread;
    *Deltasigma=pnl_vect_get(delta_sigma,12);

pnl_vect_free(&Ej);
    pnl_vect_free(&Vj);
    pnl_vect_free(&Sj);
pnl_vect_free(&sigma_ATM);
    pnl_vect_free(&delta_sigma);
pnl_mat_free(&Z1);
pnl_mat_free(&S);
pnl_mat_free(&V);
pnl_rng_free(&rng);

  return OK;
}

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


  return MCGuyon(ptMod->S0.Val.V_PDOUBLE,
  ptOpt->PayOff.Val.V_NUMFUNC_1,
  ptOpt->Maturity.Val.V_DATE - ptMod->T.Val.V_DATE,
```

```
   ptMod->sigmalower.Val.V_PDOUBLE,
 ptMod->sigmaupper.Val.V_PDOUBLE,
  ptMod->sigmadupire.Val.V_PDOUBLE,
  ptMod->nmonit.Val.V_PINT,
 Met->Par[0].Val.V_INT,Met->Par[1].Val.V_INT, Met->Par[2].Val.V_ENUM.value,
  &(Met->Res[0].Val.V_DOUBLE),
  &(Met->Res[1].Val.V_DOUBLE)
 );

}

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

  if ((strcmp(((Option *)Opt)->Name, "CallSpreadEuro") == 0))
    return OK;

  return  WRONG;
}
#endif //PremiaCurrentVersion

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

      Met->Par[0].Val.V_INT =100;
  Met->Par[1].Val.V_INT =100;
      Met->Par[2].Val.V_ENUM.value = 0;
      Met->Par[2].Val.V_ENUM.members = &PremiaEnumMCRNGs;
      Met->HelpFilenameHint = "mc_guyon";

    }
  return OK;
}

PricingMethod MET(MC_Guyon) =
{
  "MC_Guyon",
```

```
  { {"N Bins", INT2, {100}, ALLOW},
{"Multiplier", INT2, {100}, ALLOW},
    {"RandomGenerator", ENUM, {100}, ALLOW},
    {" ", PREMIA_NULLTYPE, {0}, FORBID}
  },
  CALC(MC_Guyon),
  { {"Price", DOUBLE, {100}, FORBID},
{"Forward Spread Deltasigma", DOUBLE, {100}, FORBID},
    {" ", PREMIA_NULLTYPE, {0}, FORBID}
  },
  CHK_OPT(MC_Guyon),
  CHK_mc,
  MET(Init)
};
}
```