Help

```
/* Barraquand-Martineau algorithm*/
#include <stdlib.h>
#include <stdio.h>
#include <
href../../../common/math/cdo/cdo_math_h_src.pdfmath.h>
#include <float.h>
#include <string.h>

#include "
href../../../mod/bsnd/bsnd_stdnd/bsnd_stdnd_h_src.pdfbsnd_stdnd.h"
#include "
href../../../common/math/linsys_h_src.pdfmath/linsys.h"
#include "pnl/pnl_basis.h"
#include "
href../../../mod/bsnd/bsnd_stdnd/black_h_src.pdfblack.h"
#include "
href../../../common/optype_h_src.pdfoptype.h"
#include "
href../../../common/enums_h_src.pdfenums.h"
#include "pnl/pnl_mathtools.h"
#include "
href../../../common/var_h_src.pdfvar.h"
#include "pnl/pnl_random.h"
#include "pnl/pnl_matrix.h"
#include "
href../../../common/math/mc_am_h_src.pdfmath/mc_am.h"

static double *Mesh = NULL;
static long *Weights = NULL;
static double *Path = NULL, *Mean_Cell = NULL, *Price = NULL, *Transition = NULL
static double *PathAuxPO = NULL;

static int Number_Cell(double x, int Instant, int AL_PO_Size);

static int BaMa_Allocation(int AL_PO_Size, int BS_Dimension,
                           int OP_Exercise_Dates)
{
  if (Mesh == NULL) Mesh = (double *)malloc(OP_Exercise_Dates * (AL_PO_Size + 1)
  if (Mesh == NULL) return MEMORY_ALLOCATION_FAILURE;
```

```
  if (Path == NULL) Path = (double *)malloc(OP_Exercise_Dates * BS_Dimension * s
  if (Path == NULL) return MEMORY_ALLOCATION_FAILURE;
  if (Weights == NULL) Weights = (long *)malloc(OP_Exercise_Dates * AL_PO_Size *
  if (Weights == NULL) return MEMORY_ALLOCATION_FAILURE;
  if (Mean_Cell == NULL)  Mean_Cell = (double *)malloc(OP_Exercise_Dates * AL_PO
  if (Mean_Cell == NULL) return MEMORY_ALLOCATION_FAILURE;
  if (Price == NULL)  Price = (double *)malloc(OP_Exercise_Dates * AL_PO_Size *
  if (Price == NULL) return MEMORY_ALLOCATION_FAILURE;
  if (Transition == NULL) Transition = (double *)malloc((OP_Exercise_Dates - 1)
  if (Transition == NULL) return MEMORY_ALLOCATION_FAILURE;
  if (PathAux == NULL)    PathAux = (double *)malloc(AL_PO_Size * 2 * BS_Dimensi
  if (PathAux == NULL) return MEMORY_ALLOCATION_FAILURE;
  if (PathAuxPO == NULL)  PathAuxPO = (double *)malloc((AL_PO_Size + 1) * sizeof
  if (PathAuxPO == NULL) return MEMORY_ALLOCATION_FAILURE;
  return OK;
}

static void BaMa_Liberation()
{
  if (Mesh != NULL)
    {
      free(Mesh);
      Mesh = NULL;
    }
  if (Path != NULL)
    {
      free(Path);
      Path = NULL;
    }
  if (Weights != NULL)
    {
      free(Weights);
      Weights = NULL;
    }
  if (Mean_Cell != NULL)
    {
      free(Mean_Cell);
      Mean_Cell = NULL;
    }
  if (Price != NULL)
    {
```

```
      free(Price);
      Price = NULL;
    }
  if (Transition != NULL)
    {
      free(Transition);
      Transition = NULL;
    }
  if (PathAux != NULL)
    {
      free(PathAux);
      PathAux = NULL;
    }
  if (PathAuxPO != NULL)
    {
      free(PathAuxPO);
      PathAuxPO = NULL;
    }
}

static void InitQ(NumFunc_nd *p, int AL_PO_Size, long Al_PO_Init, int BS_Dimensi
                  int OP_Exercise_Dates, double *BS_Spot, double Step, double Sq
                  int generator)
{
  int i, j, k, l;
  PnlVect VPath;
  VPath.size = BS_Dimension;
  /*mean order statistics as payoff quantizers initialization, see the documenta
  for (i = 0; i < OP_Exercise_Dates * (AL_PO_Size + 1); i++)
    Mesh[i] = 0;

  for (i = 0; i < Al_PO_Init; i++)
    {
      for (j = 0; j < AL_PO_Size; j++)
        for (k = 0; k < BS_Dimension; k++)
          PathAux[j * 2 * BS_Dimension + k] = BS_Spot[k];


      for (j = 1; j < OP_Exercise_Dates; j++)
        {
          for (k = 0; k < AL_PO_Size; k++)
```

```
            BS_Forward_Step(PathAux + k * 2 * BS_Dimension + BS_Dimension, PathA

          for (k = 1; k < AL_PO_Size + 1; k++)
            {
              VPath.array =  PathAux + (k - 1) * 2 * BS_Dimension + BS_Dimension
              PathAuxPO[k] = p->Compute(p->Par, &VPath);
            }

          Sort(AL_PO_Size, PathAuxPO);

          for (k = 1; k < AL_PO_Size + 1; k++)
            Mesh[j * (AL_PO_Size + 1) + k] += PathAuxPO[k];

          for (l = 0; l < AL_PO_Size; l++)
            for (k = 0; k < BS_Dimension; k++)
              PathAux[l * 2 * BS_Dimension + k] = PathAux[(l * 2 + 1) * BS_Dimen
        }
    }
  for (j = 1; j < OP_Exercise_Dates; j++)
    for (k = 1; k < AL_PO_Size + 1; k++)
      Mesh[j * (AL_PO_Size + 1) + k] /= (double)Al_PO_Init;


  for (j = 1; j < OP_Exercise_Dates; j++)
    {
      Mesh[j * (AL_PO_Size + 1)] = 0;
      Mesh[(j + 1) * (AL_PO_Size + 1) - 1] = DBL_MAX;
      for (k = 1; k < AL_PO_Size - 1; k++)
        Mesh[j * (AL_PO_Size + 1) + k] = (Mesh[j * (AL_PO_Size + 1) + k] +
                                          Mesh[j * (AL_PO_Size + 1) + k + 1]) *
    }
  Mesh[AL_PO_Size] = DBL_MAX;
  for (k = 0; k < AL_PO_Size; k++)
    Mesh[k] = 0;

}

static void Init_Cells(NumFunc_nd *p, int BS_Dimension, int OP_Exercise_Dates,
                       int AL_MonteCarlo_Iterations,
                       int AL_PO_Size, double *BS_Spot, double Step, double Sqrt
                       int generator)
```

```
{
  double auxop1, auxop2;
  int i, j, k, auxcell1, auxcell2;
  PnlVect VPath;
  VPath.size = BS_Dimension;

  /*computation of the payoff transition between the payoff tesselations*/
  for (i = 0; i < OP_Exercise_Dates - 1; i++)
    for (j = 0; j < AL_PO_Size; j++)
      for (k = 0; k < AL_PO_Size; k++) Transition[i * AL_PO_Size * AL_PO_Size +
  for (i = 0; i < OP_Exercise_Dates; i++)
    for (j = 0; j < AL_PO_Size; j++) Mean_Cell[i * AL_PO_Size + j] = 0;
  for (i = 0; i < OP_Exercise_Dates; i++)
    for (j = 0; j < AL_PO_Size; j++) Price[i * AL_PO_Size + j] = 0;
  for (i = 0; i < OP_Exercise_Dates; i++)
    for (j = 0; j < AL_PO_Size; j++) Weights[i * AL_PO_Size + j] = 0;

  for (k = 0; k < AL_MonteCarlo_Iterations; k++)
    {
      /*computation of a BlackScholes path*/
      ForwardPath(Path, BS_Spot, 0, OP_Exercise_Dates, BS_Dimension, Step, Sqrt_

      VPath.array = Path;
      auxop2 = p->Compute(p->Par, &VPath);
      auxcell2 = Number_Cell(auxop2, 0, AL_PO_Size);
      /*contribution of the payoff path to the transition MonteCarlo estimator*/
      for (i = 0; i < OP_Exercise_Dates - 1; i++)
        {
          auxcell1 = auxcell2;
          auxop1 = auxop2;
          VPath.array = Path + (i + 1) * BS_Dimension;
          auxop2 = p->Compute(p->Par, &VPath);
          auxcell2 = Number_Cell(auxop2, i + 1, AL_PO_Size);
          Weights[i * AL_PO_Size + auxcell1]++;
          Transition[i * AL_PO_Size * AL_PO_Size + auxcell1 * AL_PO_Size + auxce
          Mean_Cell[i * AL_PO_Size + auxcell1] += auxop1;
        }
      VPath.array = Path + (OP_Exercise_Dates - 1) * BS_Dimension;
      auxop1 = p->Compute(p->Par, &VPath);
      auxcell1 = Number_Cell(auxop1, OP_Exercise_Dates - 1, AL_PO_Size);
      Weights[(OP_Exercise_Dates - 1)*AL_PO_Size + auxcell1]++;
```

```
      Mean_Cell[(OP_Exercise_Dates - 1)*AL_PO_Size + auxcell1] += auxop1;
    }
}


static int Number_Cell(double x, int Instant, int AL_PO_Size)
{
  int min = 0, max = AL_PO_Size, j;

  /*nearest cell search*/
  do
    {
      j = (max + min) / 2;
      if (x >= Mesh[Instant * (AL_PO_Size + 1) + j])
        {
          min = j;
        }
      else
        {
          max = j;
        }
    }
  while (!((x >= Mesh[Instant * (AL_PO_Size + 1) + j]) && (x <= Mesh[Instant * (

  return j;
}


static void Close()
{
  /*memory liberation*/
  BaMa_Liberation();
  End_BS();
}

/*see the documentation for the parameters meaning*/
static int BaMa(PnlVect *BS_Spot,
                NumFunc_nd *p,
                double OP_Maturity,
                double BS_Interest_Rate,
                PnlVect *BS_Dividend_Rate,
```

```
                PnlVect *BS_Volatility,
                double *BS_Correlation,
                long AL_MonteCarlo_Iterations,
                int generator,
                int AL_PO_Size,
                int AL_PO_Init,
                int OP_Exercise_Dates,
                double *AL_BPrice)
{

  double aux, Step, Sqrt_Step, DiscountStep;
  int i, j, k, init_mc;

  int BS_Dimension = BS_Spot->size;

  /*time step*/
  Step = OP_Maturity / (double)(OP_Exercise_Dates - 1);
  Sqrt_Step = sqrt(Step);
  /*discounting factor for a time step*/
  DiscountStep = exp(-BS_Interest_Rate * Step);

  /* MC sampling */
  init_mc = pnl_rand_init(generator, BS_Dimension, AL_MonteCarlo_Iterations);

  /* Test after initialization for the generator */
  if (init_mc != OK) return init_mc;

  /*memory allocation of the BlackScholes variables*/
  Init_BS(BS_Dimension, BS_Volatility->array, BS_Correlation, BS_Interest_Rate,
  /*memory allocation of the algorithm's variables*/
  BaMa_Allocation(AL_PO_Size, BS_Dimension, OP_Exercise_Dates);
  /*initialization of the payoff quantizers*/
  InitQ(p, AL_PO_Size, AL_PO_Init, BS_Dimension, OP_Exercise_Dates, BS_Spot->arr
  /*initialization of the quantized payoff transitions*/
  Init_Cells(p, BS_Dimension, OP_Exercise_Dates, AL_MonteCarlo_Iterations, AL_PO

  /*dynamical programing prices initialization at the maturity*/
  for (k = 0; k < AL_PO_Size; k++)
    {
      if (Weights[(OP_Exercise_Dates - 1)*AL_PO_Size + k] > 0)
        {
```

```
          Price[(OP_Exercise_Dates - 1)*AL_PO_Size + k] = Mean_Cell[(OP_Exercise
        }
    }
  /*dynamical programming algorithm*/
  for (i = OP_Exercise_Dates - 2; i >= 0; i--)
    {
      for (k = 0; k < AL_PO_Size; k++)
        {
          if (Weights[i * AL_PO_Size + k] > 0)
            {
              aux = 0;
              /*conditional expectation*/
              for (j = 0; j < AL_PO_Size; j++)
                aux += Transition[i * AL_PO_Size * AL_PO_Size + k * AL_PO_Size +
              aux /= (double)Weights[i * AL_PO_Size + k];
              /*discount for a step*/
              aux *= DiscountStep;
              /*exercise decision*/
              Price[i * AL_PO_Size + k] = MAX(Mean_Cell[i * AL_PO_Size + k] / We
              /*Price[i*AL_PO_Size+k]=aux;*/
            }
        }
    }
  /*output backward price*/
  *AL_BPrice = Price[Number_Cell(p->Compute(p->Par, BS_Spot), 0, AL_PO_Size)];
  Close();
  return OK;
}

int CALC(MC_BarraquandMartineauND)(void *Opt, void *Mod, PricingMethod *Met)
{
  TYPEOPT *ptOpt = (TYPEOPT *)Opt;
  TYPEMOD *ptMod = (TYPEMOD *)Mod;
  double r;
  double *BS_cor;
  int i, res;
  PnlVect *divid = pnl_vect_create(ptMod->Size.Val.V_PINT);
  PnlVect *spot, *sig;

  spot = ptMod->S0.Val.V_PNLVECT;
  sig = ptMod->Sigma.Val.V_PNLVECT;
```

```
  for (i = 0; i < ptMod->Size.Val.V_PINT; i++)
    pnl_vect_set(divid, i,
                 log(1. + GET(ptMod->Divid.Val.V_PNLVECT, i) / 100.));

  r = log(1. + ptMod->R.Val.V_DOUBLE / 100.);

  if ((BS_cor = malloc(ptMod->Size.Val.V_PINT * ptMod->Size.Val.V_PINT * sizeof(
    return MEMORY_ALLOCATION_FAILURE;
  for (i = 0; i < ptMod->Size.Val.V_PINT * ptMod->Size.Val.V_PINT; i++)
    BS_cor[i] = ptMod->Rho.Val.V_DOUBLE;
  for (i = 0; i < ptMod->Size.Val.V_PINT; i++)
    BS_cor[i * ptMod->Size.Val.V_PINT + i] = 1.0;

  res = BaMa(spot,
             ptOpt->PayOff.Val.V_NUMFUNC_ND,
             ptOpt->Maturity.Val.V_DATE - ptMod->T.Val.V_DATE,
             r, divid, sig,
             BS_cor,
             Met->Par[0].Val.V_LONG,
             Met->Par[1].Val.V_ENUM.value,
             Met->Par[2].Val.V_INT,
             Met->Par[3].Val.V_INT,
             Met->Par[4].Val.V_INT,
             &(Met->Res[0].Val.V_DOUBLE));
  pnl_vect_free(&divid);
  free(BS_cor);

  return res;
}

static int CHK_OPT(MC_BarraquandMartineauND)(void *Opt, void *Mod)
{
  Option *ptOpt = (Option *)Opt;
  TYPEOPT *opt = (TYPEOPT *)(ptOpt->TypeOpt);

  if ((opt->EuOrAm).Val.V_BOOL == AMER)
    return OK;
  return  WRONG;
}
```

```
static int MET(Init)(PricingMethod *Met, Option *Opt)
{
  if (Met->init == 0)
    {
      Met->init = 1;
      Met->Par[0].Val.V_LONG = 50000;
      Met->Par[1].Val.V_ENUM.value = 0;
      Met->Par[1].Val.V_ENUM.members = &PremiaEnumMCRNGs;
      Met->Par[2].Val.V_INT = 200;
      Met->Par[3].Val.V_INT = 100;
      Met->Par[4].Val.V_INT = 10;
    }
  return OK;
}

PricingMethod MET(MC_BarraquandMartineauND) =
{
  "MC_BarraquandMartineau_ND",
  { {"N iterations", LONG, {100}, ALLOW},
    {"RandomGenerator", ENUM, {100}, ALLOW},
    {"Number of Cells", INT, {100}, ALLOW},
    {"Size of grid initialising sample", INT, {100}, ALLOW},
    {"Number of Exercise Dates", INT, {100}, ALLOW},
    {" ", PREMIA_NULLTYPE, {0}, FORBID}
  },
  CALC(MC_BarraquandMartineauND),
  { {"Price", DOUBLE, {100}, FORBID},
    {" ", PREMIA_NULLTYPE, {0}, FORBID}
  },
  CHK_OPT(MC_BarraquandMartineauND),
  CHK_mc,
  MET(Init)
};
```