Help

```
/* Random Quantization Algorithm */
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <float.h>

#include "bsnd_stdnd.h"
#include "math/linsys.h"
#include "pnl/pnl_basis.h"
#include "black.h"
#include "optype.h"
#include "var.h"
#include "enums.h"
#include "pnl/pnl_random.h"
#include "premia_obj.h"
#include "pnl/pnl_matrix.h"

/* epsilon to detect if continuation value is reached */
#define EPS_CONT 0.0000001

static double *Q = NULL, *BSQ = NULL, *Weights = NULL, *Trans = NULL, *Price = N
static double *Aux_B = NULL, *Brownian_Bridge = NULL;
static int *Number_Cell = NULL;

static int (*Search_Method)(double *S, int Time, int AL_T_Size, long OP_EmBS_Di,

static int RaQ_Allocation(int AL_T_Size, int BS_Dimension,
                          int OP_Exercice_Dates, long AL_MonteCarlo_Iterations)
{
  if (Tesselation == NULL)
    Tesselation = (double *)malloc(AL_T_Size * BS_Dimension * sizeof(double));
  if (Tesselation == NULL) return MEMORY_ALLOCATION_FAILURE;
  if (Q == NULL)
    Q = (double *)malloc(AL_T_Size * OP_Exercice_Dates * BS_Dimension * sizeof(d
  if (Q == NULL) return MEMORY_ALLOCATION_FAILURE;
  if (BSQ == NULL)
    BSQ = (double *)malloc(AL_T_Size * OP_Exercice_Dates * BS_Dimension * sizeof
  if (BSQ == NULL) return MEMORY_ALLOCATION_FAILURE;
  if (Brownian_Bridge == NULL)
```

```
    Brownian_Bridge = (double *)malloc(AL_MonteCarlo_Iterations * BS_Dimension *
  if (Brownian_Bridge == NULL) return MEMORY_ALLOCATION_FAILURE;
  if (Number_Cell == NULL)
    Number_Cell = (int *)malloc(AL_MonteCarlo_Iterations * sizeof(int));
  if (Number_Cell == NULL) return MEMORY_ALLOCATION_FAILURE;
  if (Trans == NULL)
    Trans = (double *)malloc(AL_T_Size * AL_T_Size * sizeof(double));
  if (Trans == NULL) return MEMORY_ALLOCATION_FAILURE;
  if (Weights == NULL)
    Weights = (double *)malloc(AL_T_Size * sizeof(double));
  if (Weights == NULL) return MEMORY_ALLOCATION_FAILURE;
  if (Price == NULL)
    Price = (double *)malloc(OP_Exercice_Dates * AL_T_Size * sizeof(double));
  if (Price == NULL) return MEMORY_ALLOCATION_FAILURE;
  if (Aux_B == NULL)
    Aux_B = (double *)malloc(BS_Dimension * sizeof(double));
  if (Aux_B == NULL) return MEMORY_ALLOCATION_FAILURE;
  return OK;
}

static void RaQ_Liberation()
{
  if (Brownian_Bridge != NULL)
    {
      free(Brownian_Bridge);
      Brownian_Bridge = NULL;
    }
  if (Tesselation != NULL)
    {
      free(Tesselation);
      Tesselation = NULL;
    }
  if (Q != NULL)
    {
      free(Q);
      Q = NULL;
    }
  if (BSQ != NULL)
    {
      free(BSQ);
      BSQ = NULL;
```

```
    }
  if (Trans != NULL)
    {
      free(Trans);
      Trans = NULL;
    }
  if (Weights != NULL)
    {
      free(Weights);
      Weights = NULL;
    }
  if (Price != NULL)
    {
      free(Price);
      Price = NULL;
    }
  if (Aux_B != NULL)
    {
      free(Aux_B);
      Aux_B = NULL;
    }
  if (Number_Cell != NULL)
    {
      free(Number_Cell);
      Number_Cell = NULL;
    }
}

static int NearestCellD1(double *S, int Time, int AL_T_Size, long OP_ExmBS_Di, i
{
  int j, l = 0;
  long ind;
  double min = DBL_MAX, aux;

  /*computation of the nearest cell index of the vector S with respect to the on
  ind = Time;
  for (j = 0; j < AL_T_Size; j++)
    {
      aux = fabs(*S - Q[ind]);
      ind += OP_ExmBS_Di;
      if (min > aux)
```

```
        {
          min = aux;
          l = j;
        }
    }
  return l;
}

static int NearestCell(double *S, int Time, int AL_T_Size, long OP_ExmBS_Di, int
{
  int j, k, l = 0;
  long ind, TimemBS_Dimension = Time * BS_Dimension;
  double  min = DBL_MAX, aux, auxnorm;

  /*computation of the nearest cell index of the vector S with respect to the te

  ind = TimemBS_Dimension;
  for (j = 0; j < AL_T_Size; j++)
    {
      aux = 0;
      k = 0;
      while ((aux < min) && (k < BS_Dimension))
        {
          auxnorm = S[k] - Q[ind + k];
          aux += auxnorm * auxnorm;
          k++;
        }
      ind += OP_ExmBS_Di;
      if (aux < min)
        {
          min = aux;
          l = j;
        }
    }

  return l;
}

static void InitTesselation(int BS_Dimension, int AL_T_Size, int OP_Exercice_Dat
{
  int i, j, k;
```

```
  for (i = 0; i < AL_T_Size; i++)
    {
      for (k = 0; k < BS_Dimension; k++)
        {
          Q[i * OP_Exercice_Dates * BS_Dimension + k] = 0.;
        }
    }
  /*initialisation of the brownian quantizer (Q) and of the of the BlackScholes
  for (j = 1; j < OP_Exercice_Dates; j++)
    {
      for (i = 0; i < AL_T_Size; i++)
        {
          for (k = 0; k < BS_Dimension; k++)
            {
              /*brownian motion increment*/
              Q[i * OP_Exercice_Dates * BS_Dimension + j * BS_Dimension + k] = Q
            }
          /*BlackScholes stock computation related to a brownian motion value*/
          BlackScholes_Transformation((double)j * Step, BSQ + i * OP_Exercice_Da
        }
    }
}

static void Compute_Transition(long AL_MonteCarlo_Iterations, int AL_T_Size, int
                               int OP_EmBS_Di, int OP_Exercice_Dates, int Time)
{
  int i, j, AuxNumCell;
  long l;

  for (i = 0; i < AL_T_Size; i++)
    {
      Weights[i] = 0.;
      for (j = 0; j < AL_T_Size; j++)
        {
          Trans[i * AL_T_Size + j] = 0.;
        }
    }
  /*computation of the brownian bridge transition probabilities from the quantiz
  for (l = 0; l < AL_MonteCarlo_Iterations; l++)
    {
```

```
      AuxNumCell = Search_Method(Brownian_Bridge + l * BS_Dimension, Time, AL_T_
      Trans[AuxNumCell * AL_T_Size + Number_Cell[l]] += 1.;
      Weights[AuxNumCell] += 1.;
      Number_Cell[l] = AuxNumCell;
    }
  /*normalization*/
  for (i = 0; i < AL_T_Size; i++)
    {
      for (j = 0; j < AL_T_Size; j++)
        {
          if (Weights[i] > 0.)
            {
              Trans[i * AL_T_Size + j] /= Weights[i];
            }
        }
    }
}

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

/*see the documentation for the parameters meaning*/
static int  RaQ(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 OP_Exercice_Dates,
                int AL_T_Size,
                double *AL_BPrice,
                double *AL_FPrice)
{
  int i, j, k, AuxNumCell, ret, init_mc;
```

```
long l;
int BS_Dimension = BS_Spot->size;
long OP_ExmBS_Di = (long)OP_Exercice_Dates * BS_Dimension;
double Step, Sqrt_Step, DiscountStep, aux;
PnlVect VStock;
VStock.size = BS_Dimension;

/* 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;

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

/*memory allocation of the BlackScholes variables*/
ret = Init_BS(BS_Dimension, BS_Volatility->array,
              BS_Correlation, BS_Interest_Rate, BS_Dividend_Rate->array);
if (ret != OK) return ret;

/*memory allocation of the algorithm's variables*/
ret = RaQ_Allocation(AL_T_Size, BS_Dimension, OP_Exercice_Dates, AL_MonteCarlo
if (ret != OK) return ret;

if (BS_Dimension == 1)
  /*fast nearest cell search for the dimension one*/
  Search_Method = NearestCellD1;
else
  Search_Method = NearestCell;

/*initialization of the random quantizers of the brownian bridge*/
InitTesselation(BS_Dimension, AL_T_Size, OP_Exercice_Dates, Step, BS_Spot->arr
                Sqrt_Step, generator);
/*initialization of the brownian bridge at the maturity*/
Init_Brownian_Bridge(Brownian_Bridge, AL_MonteCarlo_Iterations, BS_Dimension,
/*initialisation of the dynamical programming prices at the maturity*/
for (i = 0; i < AL_T_Size; i++)
```

```
    {
      VStock.array = BSQ + i * OP_Exercice_Dates * BS_Dimension + (OP_Exercice_D
      Price[(OP_Exercice_Dates - 1)*AL_T_Size + i] = p->Compute(p->Par, &VStock)
    }
  /*quantization of the brownian bridge*/
  for (i = 0; i < AL_MonteCarlo_Iterations; i++)
    {
      Number_Cell[i] = Search_Method(Brownian_Bridge + i * BS_Dimension, OP_Exer
    }
  /*dynamical programming algorithm*/
  for (i = OP_Exercice_Dates - 2; i >= 1; i--)
    {
      /*computation of the brownian bridge at time i*/
      Compute_Brownian_Bridge(Brownian_Bridge, i * Step, Step, BS_Dimension, AL_
      /*computation of the quantized transition kernel of the brownian bridge be
      Compute_Transition(AL_MonteCarlo_Iterations, AL_T_Size, BS_Dimension, OP_E
      /*approximation of the conditionnal expectations*/
      for (j = 0; j < AL_T_Size; j++)
        {
          aux = 0;
          for (k = 0; k < AL_T_Size; k++)
            {
              aux += Price[(i + 1) * AL_T_Size + k] * Trans[j * AL_T_Size + k];
            }
          /*discounting for a time step*/
          aux *= DiscountStep;
          /*aux contains the continuation value at quantization point j and time
          /*exercise decision*/
          VStock.array = BSQ + j * OP_Exercice_Dates * BS_Dimension + (i) * BS_D
          Price[i * AL_T_Size + j] = MAX(p->Compute(p->Par, &VStock), aux);
        }
    }

  aux = 0;
  for (k = 0; k < AL_T_Size; k++)
    {
      aux += (Price[AL_T_Size + k]) * Weights[k];
    }

  aux /= (double)AL_MonteCarlo_Iterations;
  aux *= DiscountStep;
```

```
/*output backward price*/
*AL_BPrice = MAX(p->Compute(p->Par, BS_Spot), aux);
*AL_FPrice = 0;

/*  Forward price */
if (*AL_BPrice == p->Compute(p->Par, BS_Spot))
  {
    *AL_FPrice = *AL_BPrice;
  }
else
  {
    for (l = 0; l < AL_MonteCarlo_Iterations; l++)
      {
        /*spot of the brownian motion*/
        for (k = 0; k < BS_Dimension; k++)
          {
            Aux_B[k] = 0.;
          }
        i = 0;
        /*optimal stopping for a quantized path*/
        do
          {
            i++;
            for (k = 0; k < BS_Dimension; k++)
              {
                Aux_B[k] += Sqrt_Step * pnl_rand_normal(generator);
              }
            /*search of the Aux_B number cell*/
            AuxNumCell = Search_Method(Aux_B, i, AL_T_Size, OP_Exercice_Dates
            VStock.array = BSQ + AuxNumCell * OP_Exercice_Dates * BS_Dimension
          }
        while (p->Compute(p->Par, &VStock) < Price[i * AL_T_Size + AuxNumCell]
        /*MonteCarlo formulae for the forward price*/
        VStock.array = BSQ + AuxNumCell * OP_Exercice_Dates * BS_Dimension + i
        *AL_FPrice += Discount((double)i * Step, BS_Interest_Rate) * p->Comput
      }
    /*output forward price*/
    *AL_FPrice /= (double)AL_MonteCarlo_Iterations;
  }
```

```
  Close();
  return OK;
}


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

  spot = pnl_vect_compact_to_pnl_vect(ptMod->S0.Val.V_PNLVECTCOMPACT);
  sig = pnl_vect_compact_to_pnl_vect(ptMod->Sigma.Val.V_PNLVECTCOMPACT);

  for (i = 0; i < ptMod->Size.Val.V_PINT; i++)
    pnl_vect_set(divid, i,
                 log(1. + pnl_vect_compact_get(ptMod->Divid.Val.V_PNLVECTCOMPACT

  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 = RaQ(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->Res[0].Val.V_DOUBLE),
            &(Met->Res[1].Val.V_DOUBLE));
  pnl_vect_free(&divid);
  pnl_vect_free(&spot);
  pnl_vect_free(&sig);
  free(BS_cor);

  return res;
}


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

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

static int MET(Init)(PricingMethod *Met, Option *Opt)
{
  if (Met->init == 0)
    {
      Met->init = 1;
      Met->Par[0].Val.V_LONG = 10000;
      Met->Par[1].Val.V_ENUM.value = 0;
      Met->Par[1].Val.V_ENUM.members = &PremiaEnumMCRNGs;
      Met->Par[2].Val.V_INT = 10;
      Met->Par[3].Val.V_INT = 200;
    }
  return OK;
}

PricingMethod MET(MC_RandomQuantizationND) =
{
  "MC_Random_Quantization_nd",
  { {"N iterations", LONG, {100}, ALLOW},
    {"RandomGenerator", ENUM, {100}, ALLOW},
```

```
    {"Number of Exercise Dates", INT, {100}, ALLOW},
    {"Tesselation Size", INT, {100}, ALLOW},
    {" ", PREMIA_NULLTYPE, {0}, FORBID}
  },
  CALC(MC_RandomQuantizationND),
  { {"Forward Price", DOUBLE, {100}, FORBID}, {"Backward Price", DOUBLE, {100},
    {" ", PREMIA_NULLTYPE, {0}, FORBID}
  },
  CHK_OPT(MC_RandomQuantizationND),
  CHK_mc,
  MET(Init)
};

#undef EPS_CONT
```