Help
```
/* Optimal 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 "enums.h"
#include "var.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 const double LandMarkNorm=30;

static double *Q=NULL, *BSQ=NULL, *Weights=NULL, *Trans=NUL
    L, *Price=NULL, *Tesselation=NULL;
static double *Aux_B=NULL, *Aux_BN=NULL, *Brownian_Bridge=
    NULL, *Cells_To_LandMark_Up=NULL, *Cells_To_LandMark_Do=NUL
    L;
static double *Radius=NULL, *Dist_To_LandMark=NULL, *Sqrt_
    Inv_Time=NULL;
static int *Number_Cell=NULL;

static int (*Search_Method)(double *S,int Time, int AL_T_Si
    ze, long OP_EmBS_Di, int BS_Dimension);

static int QOpt_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 (Radius==NULL) Radius=(double*)malloc(AL_T_Size*sizeof
  (double));
if (Radius==NULL) return MEMORY_ALLOCATION_FAILURE;
if (Dist_To_LandMark==NULL) Dist_To_LandMark=(double*)mal
  loc(AL_T_Size*sizeof(double));
if (Dist_To_LandMark==NULL) return MEMORY_ALLOCATION_FAI
  LURE;
if (Cells_To_LandMark_Up==NULL)
  Cells_To_LandMark_Up=(double*)malloc(BS_Dimension*AL_T_
  Size*sizeof(double));
if (Cells_To_LandMark_Up==NULL) return MEMORY_ALLOCATION_
  FAILURE;
if (Cells_To_LandMark_Do==NULL)
  Cells_To_LandMark_Do=(double*)malloc(BS_Dimension*AL_T_
  Size*sizeof(double));
if (Cells_To_LandMark_Do==NULL) return MEMORY_ALLOCATION_
  FAILURE;
if (Q==NULL)
  Q=(double*)malloc(AL_T_Size*OP_Exercice_Dates*BS_Dimens
  ion*sizeof(double));
if (Q==NULL) return MEMORY_ALLOCATION_FAILURE;
if (BSQ==NULL)
  BSQ=(double*)malloc(AL_T_Size*OP_Exercice_Dates*BS_Dim
  ension*sizeof(double));
if (BSQ==NULL) return MEMORY_ALLOCATION_FAILURE;
if (Brownian_Bridge==NULL)
  Brownian_Bridge=(double*)malloc(AL_MonteCarlo_Iteratio
  ns*BS_Dimension*sizeof(double));
if (Brownian_Bridge==NULL) return MEMORY_ALLOCATION_FAILU
  RE;
if (Number_Cell==NULL)
  Number_Cell=(int*)malloc(AL_MonteCarlo_Iterations*size
  of(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*size
    of(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;
  if (Aux_BN==NULL)
    Aux_BN=(double*)malloc(BS_Dimension*sizeof(double));
  if (Aux_BN==NULL) return MEMORY_ALLOCATION_FAILURE;
  if (Sqrt_Inv_Time==NULL)
    Sqrt_Inv_Time=(double*)malloc(OP_Exercice_Dates*sizeof(
    double));
  if (Sqrt_Inv_Time==NULL) return MEMORY_ALLOCATION_FAILU
    RE;
  return OK;
}

static void QOpt_Liberation()
{
  if (Brownian_Bridge!=NULL) { free(Brownian_Bridge);  Brow
    nian_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 (Aux_BN!=NULL) { free(Aux_BN); Aux_BN=NULL; }
  if (Radius!=NULL) { free(Radius); Radius=NULL;}
  if (Sqrt_Inv_Time!=NULL) { free(Sqrt_Inv_Time); Sqrt_Inv_
    Time=NULL;}
  if (Cells_To_LandMark_Up!=NULL) {free(Cells_To_LandMark_
    Up);Cells_To_LandMark_Up=NULL; }
  if (Cells_To_LandMark_Do!=NULL) {free(Cells_To_LandMark_
    Do);Cells_To_LandMark_Do=NULL; }
  if (Dist_To_LandMark!=NULL) {free(Dist_To_LandMark);Dist_
    To_LandMark=NULL;}
```

```
  if (Number_Cell!=NULL) {free(Number_Cell);Number_Cell=NUL
    L;}
}

static int NearestCellD1(double *S,int Time, int AL_T_Size,
     long OP_EmBS_Di, int BS_Dimension)
{
  int j,l=0;
  long ind;
  double min=DBL_MAX,aux;
  /*one dimensional nearest cell search*/
  ind=0;
  for (j=0;j<AL_T_Size;j++)
    {
      aux=fabs(*S-Tesselation[ind]);
      ind+=BS_Dimension;
      if (min>aux) { min=aux;  l=j;}
    }
  return l;
}

static int NearestCell(double *S,int Time, int AL_T_Size,
    long OP_EmBS_Di, int BS_Dimension)
{
  /*the Brownian Motion S must be normalised. */
  /*used if the fast nearest cell search fails; this event
    is of very small probability*/

  int j,k,l=0;
  double  min=DBL_MAX,aux,auxnorm;


  for (j=0;j<AL_T_Size;j++)
    {
      aux=0;
      k=0;

      while ((aux<min)&&(k<BS_Dimension))
        {
          auxnorm=S[k]-Tesselation[j*BS_Dimension+k];
          aux+=auxnorm*auxnorm;
```

```
        k++;
      }

    if (aux<min){ min=aux; l=j;}
  }

  return l;
}

static int FastNearestCell(double *S,int Time, int AL_T_Si
    ze, long OP_EmBS_Di, int BS_Dimension)
{
  /* The Brownian Motion S must be normalised. */
  /* see the documentation for the explanation of the fast
    nearest cell search*/
  int j,k,l=-1,m;
  int InegTrue;
  long jmBS_Dim,mmAL_T_Sizepj;
  double  min=DBL_MAX,aux,auxnorm;



  for (m=0;m<BS_Dimension;m++){
    Dist_To_LandMark[m]=0;
    for (k=0;k<m;k++){
      Dist_To_LandMark[m]+=S[k]*S[k];
    }
    Dist_To_LandMark[m]+=(S[m]-LandMarkNorm)*(S[m]-LandMar
    kNorm);
    for (k=m+1;k<BS_Dimension;k++){
      Dist_To_LandMark[m]+=S[k]*S[k];
    }
  }
  jmBS_Dim=0;
  for (j=0;j<AL_T_Size;j++){
    aux=0;
    k=0;
    m=0;
    InegTrue=1;
    mmAL_T_Sizepj=j;
    do {
```

```
      InegTrue=(InegTrue)&&(Dist_To_LandMark[m]<=Cells_To_
    LandMark_Up[mmAL_T_Sizepj])&&(Dist_To_LandMark[m]>=Cells_To_
    LandMark_Do[mmAL_T_Sizepj]);
      m++;
      mmAL_T_Sizepj+=AL_T_Size;
    } while ((InegTrue)&&(m<BS_Dimension));

    if (InegTrue){
      while ((aux<min)&&(k<BS_Dimension)){
        auxnorm=S[k]-Tesselation[jmBS_Dim+k];
        aux+=auxnorm*auxnorm;
        k++;
      }
      if (aux<min){
        min=aux;
        l=j;
      }
    }
    jmBS_Dim+=BS_Dimension;
  }
  if (l==-1){
    l=NearestCell(S,Time,AL_T_Size,OP_EmBS_Di,BS_Dimension)
    ;
    return l;
  }
  return l;
}

static int InitTesselation(char *path, char *name, int BS_
    Dimension, int AL_T_Size)
{
  int i,j,nvl;
  char NameDef[MAX_PATH_LEN];
  FILE *filtes;
  /*load of a file containing an optimal tesselation*/
  NameDef[0]='{0';

  if (*name=='d'){
    sprintf(NameDef,"%sd%dn%d.tes",path,BS_Dimension,AL_T_
    Size);
    filtes=fopen(NameDef,"r");
```

```
    if (filtes==NULL) return UNABLE_TO_OPEN_FILE;
  } else {
    sprintf(NameDef,"%s%s",path,name);
    filtes=fopen(NameDef,"r");
    if (filtes==NULL) return UNABLE_TO_OPEN_FILE;
  }
  if (BS_Dimension!=1){
    nvl=fscanf(filtes,"%s{n",NameDef);
    nvl=fscanf(filtes,"%s{n",NameDef);
    nvl=fscanf(filtes,"%s{n",NameDef);
    nvl=fscanf(filtes,"%s{n",NameDef);
  } else {
    nvl=fscanf(filtes,"%s{n",NameDef);
    nvl=fscanf(filtes,"%s{n",NameDef);
    nvl=fscanf(filtes,"%s{n",NameDef);
  }
  for (i=0;i<AL_T_Size;i++){
    for(j=0;j<BS_Dimension;j++){
      nvl=fscanf(filtes,"%lf ",Tesselation+i*BS_Dimension+
    j);
      if (nvl!=1) return BAD_TESSELATION_FORMAT;
    }
    if (BS_Dimension!=1){
      nvl=fscanf(filtes,"%lf ",Radius+i);
      if (nvl!=1) return BAD_TESSELATION_FORMAT;
    }
  }
  fclose(filtes);
  return OK;
}

static void InitLandMark(int BS_Dimension, int AL_T_Size)
{
  int i,j,k;
  double AuxD;

  /*initialization of the distances (landmarks,quantization
    s points)+/-(cell radius) used in the fast nearest cell se
    arch*/
  for (i=0;i<BS_Dimension;i++){
    for (j=0;j<AL_T_Size;j++){
```

```
      AuxD=0.;
      for (k=0;k<i;k++){
        AuxD+=Tesselation[j*BS_Dimension+k]*Tesselation[j*
    BS_Dimension+k];
      }
      AuxD+=(Tesselation[j*BS_Dimension+i]-LandMarkNorm)*(
    Tesselation[j*BS_Dimension+i]-LandMarkNorm);
      for (k=i+1;k<BS_Dimension;k++){
        AuxD+=Tesselation[j*BS_Dimension+k]*Tesselation[j*
    BS_Dimension+k];
      }
      AuxD=sqrt(AuxD);
      Cells_To_LandMark_Up[i*AL_T_Size+j]=(AuxD+Radius[j])*
    (AuxD+Radius[j]);
      Cells_To_LandMark_Do[i*AL_T_Size+j]=(AuxD-Radius[j])*
    (AuxD-Radius[j]);
    }
  }
}

static void Tesselations_Scale(int AL_T_Size, int OP_Exerc
    ice_Dates, int BS_Dimension,
                               double Step, double *BS_Spo
    t)
{
  int i,j,k;
  double SqrtTime;
  /*scalings of the initial N(0,Id) optimal tesselation*/
  for (j=1;j<OP_Exercice_Dates;j++){
    SqrtTime=sqrt((double)j*Step);
    for (i=0;i<AL_T_Size;i++){
      for (k=0;k<BS_Dimension;k++){
        Q[i*OP_Exercice_Dates*BS_Dimension+j*BS_Dimension+
    k]=SqrtTime*Tesselation[i*BS_Dimension+k];
      }
      BlackScholes_Transformation((double)j*Step,BSQ+i*OP_
    Exercice_Dates*BS_Dimension+j*BS_Dimension,Q+i*OP_Exercice_
    Dates*BS_Dimension+j*BS_Dimension,BS_Dimension,BS_Spot);
    }
  }
}
```

```
static void
Compute_Transition(long AL_MonteCarlo_Iterations, int AL_T_
    Size, int BS_Dimension,
                   int OP_EmBS_Di, int OP_Exercice_Dates,
    int Time, double Sqrt_Inv_Time)
{
  int i,j,k,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 probabil
    ities from the quantizations cells at time "Time" to the qu
    antization cells at time "Time+1" (see the documantation fo
    r more informations)*/
  for (l=0;l<AL_MonteCarlo_Iterations;l++){
    for (k=0;k<BS_Dimension;k++){
      Aux_B[k]=Sqrt_Inv_Time*Brownian_Bridge[l*BS_Dimensio
    n+k];
    }
    AuxNumCell=Search_Method(Aux_B,Time,AL_T_Size,OP_EmBS_
    Di,BS_Dimension);

    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*/
  QOpt_Liberation();
  End_BS();
}

/*see the documentation for the parameters meaning*/
static int QOpt(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,
                char *AL_Tesselation_Path,
                char *AL_Tesselation_Name,
                double *AL_FPrice,
                double *AL_BPrice)
{
  int i,j,k,AuxNumCell, init_mc, init;
  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*/
init=Init_BS(BS_Dimension,BS_Volatility->array,
             BS_Correlation,BS_Interest_Rate,BS_Dividend_
  Rate->array);
if (init!=OK) return init;
/*memory allocation of the algorithm's variables*/
init=QOpt_Allocation(AL_T_Size,BS_Dimension,OP_Exercice_
  Dates,AL_MonteCarlo_Iterations);
if (init!=OK) return init;

if (BS_Dimension==1) Search_Method=NearestCellD1;
else Search_Method=FastNearestCell;

for (j=1;j<OP_Exercice_Dates;j++)
  Sqrt_Inv_Time[j]=1./sqrt((double)j*Step);

/*initialization of the optimal quantizers*/
init=InitTesselation(AL_Tesselation_Path,AL_Tesselation_
  Name,BS_Dimension,AL_T_Size);
if (init!=OK) return init;
/*landmarks initialization for the fast nearest cell sear
  ch procedure*/
if (BS_Dimension>1)
  InitLandMark(BS_Dimension,AL_T_Size);

/*optimal tesselation scalings*/
Tesselations_Scale(AL_T_Size,OP_Exercice_Dates,BS_Dimens
  ion,Step,BS_Spot->array);
/*initialization of the brownian bridge at the maturity*/
Init_Brownian_Bridge(Brownian_Bridge,AL_MonteCarlo_Itera
  tions,BS_Dimension,OP_Maturity, generator);
/*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_Dates-1)*BS_Dimension;
```

```
    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++){
  /*normalisation for the nearest cell search procedure*/
  for (k=0;k<BS_Dimension;k++){
    Aux_B[k]=Sqrt_Inv_Time[OP_Exercice_Dates-1]*Brownian_
  Bridge[i*BS_Dimension+k];
  }
  /*nearest cell search*/
  Number_Cell[i]=Search_Method(Aux_B,OP_Exercice_Dates-1,
  AL_T_Size,OP_ExmBS_Di,BS_Dimension);
}
/*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_MonteCarlo_Iterations, generator);
  /*computation of the quantized transition kernel of th
  e brownian bridge between times i and i+1*/
  Compute_Transition(AL_MonteCarlo_Iterations,AL_T_Size,
  BS_Dimension,OP_ExmBS_Di,OP_Exercice_Dates,i,Sqrt_Inv_Time[
  i]);
  /*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 i*/
    /*exercise decision*/
    VStock.array=BSQ+j*OP_Exercice_Dates*BS_Dimension+i*
  BS_Dimension;
    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);

/*  Forward price */
*AL_FPrice = 0.;
if (*AL_BPrice==p->Compute(p->Par,BS_Spot)){
  *AL_FPrice=*AL_BPrice;
}
else {
  double payoff;
  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);
            /*normalization of Aux_B*/
            Aux_BN[k]=Sqrt_Inv_Time[i]*Aux_B[k];
          }
        /*search of the Aux_BN number cell*/
        AuxNumCell=Search_Method(Aux_BN,i,AL_T_Size,OP_
  Exercice_Dates*BS_Dimension,BS_Dimension);
        VStock.array = BSQ+AuxNumCell*OP_Exercice_Dates*
  BS_Dimension+i*BS_Dimension;
        payoff = p->Compute(p->Par, &VStock);
      } while (payoff<Price[i*AL_T_Size+AuxNumCell] - EP
  S_CONT);
    /*MonteCarlo formulae for the forward price*/
    VStock.array=BSQ+AuxNumCell*OP_Exercice_Dates*BS_Dim
```

```
    ension+i*BS_Dimension;
      *AL_FPrice+=Discount(i*Step,BS_Interest_Rate)*(p->
    Compute(p->Par, &VStock));
    }
    /*output forward price*/
    *AL_FPrice/=(double)AL_MonteCarlo_Iterations;
  }
  Close();
  return OK;
}


int CALC(MC_QuantizationND)(void *Opt, void *Mod, Pricing
    Method *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);
  char tes[MAX_PATH_LEN];
  char *file ="d";
  PnlVect *spot, *sig;

  spot = pnl_vect_compact_to_pnl_vect (ptMod->S0.Val.V_PNLV
    ECTCOMPACT);
  sig = pnl_vect_compact_to_pnl_vect (ptMod->Sigma.Val.V_PN
    LVECTCOMPACT);

  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, 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(double)))==NULL)
    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;

  /* path name for the tesselation file*/
  strcpy(tes, premia_data_dir);
  strcat(tes, "/"); strcat(tes,"tes"); strcat(tes, "/");


  res=QOpt(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,
           tes,
           file,
           &(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_QuantizationND)(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_QuantizationND)=
{
  "MC_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_QuantizationND),
  {{"Forward Price",DOUBLE,{100},FORBID},{"Backward Price",
    DOUBLE,{100},FORBID},
   {" ",PREMIA_NULLTYPE,{0},FORBID}},
  CHK_OPT(MC_QuantizationND),
  CHK_mc,
  MET(Init)
};

#undef EPS_CONT
```

References