Help

```
/* Black-Scholes model */
#include <stdlib.h>
#include <
href../../../common/math/cdo/cdo_math_h_src.pdfmath.h>
#include <string.h>

#include "pnl/pnl_random.h"
#include "
href../../../common/optype_h_src.pdfoptype.h"
#include "pnl/pnl_mathtools.h"
#include "
href../../../mod/bsnd/bsnd_stdnd/black_h_src.pdfblack.h"
#include "
href../../../mod/bsnd/bsnd_stdnd/bsnd_stdnd_h_src.pdfbsnd_stdnd.h"
#include "pnl/pnl_cdf.h"

static double *VDSC = NULL, *AuxBS = NULL;
static double *Vector_BS_Mean = NULL, *AuxDBS = NULL;
static double *Aux_Stock = NULL;
static PnlMat *Inv_Sqrt_BS_Dispersion = NULL;
static PnlMat *Sigma = NULL, *InvSigma = NULL;
static double *Aux_BS_TD_1 = NULL, *Aux_BS_TD_2 = NULL;
static double DetInvSigma, Norm_BS_TD;


int Init_BS(int BS_Dimension, double *BS_Volatility,
            double *BS_Correlation, double BS_Interest_Rate,
            double *BS_Dividend_Rate)
{
  int i, j, k;
  double aux;

  if (Sigma == NULL)
    {
      Sigma = pnl_mat_create(BS_Dimension, BS_Dimension);
      if (Sigma == NULL) return MEMORY_ALLOCATION_FAILURE;
      /*BlackSholes dispersion matrix*/
      for (i = 0; i < BS_Dimension; i++)
        {
```

```
    pnl_mat_set(Sigma, i, i, BS_Volatility[i]*BS_Volatility[i]);
    for (j = i + 1; j < BS_Dimension; j++)
      {
        double tmp;
        tmp = BS_Volatility[i] * BS_Volatility[j] * BS_Correlation[i * BS_
        pnl_mat_set(Sigma, i, j, tmp);
        pnl_mat_set(Sigma, j, i, tmp);
      }
  }
/*square root of the BlackSholes dispersion matrix*/
pnl_mat_chol(Sigma);

if (VDSC == NULL)
  {
    VDSC = (double *)malloc(BS_Dimension * sizeof(double));
    if (VDSC == NULL)
      return MEMORY_ALLOCATION_FAILURE;
    for (i = 0; i < BS_Dimension; i++)
      {
        aux = 0;
        for (j = 0; j <= i; j++)
          aux += pnl_mat_get(Sigma, i, j) * pnl_mat_get(Sigma, i, j);
        VDSC[i] = aux * 0.5;
      }
  }

if (AuxBS == NULL)
  {
    AuxBS = (double *)malloc(BS_Dimension * sizeof(double));
    if (AuxBS == NULL)
      return MEMORY_ALLOCATION_FAILURE;
    for (i = 0; i < BS_Dimension; i++)
      AuxBS[i] = VDSC[i] - BS_Interest_Rate + BS_Dividend_Rate[i];
  }


if (Aux_Stock == NULL)
  {
    Aux_Stock = (double *)malloc(BS_Dimension * sizeof(double));
    if (Aux_Stock == NULL)
      return MEMORY_ALLOCATION_FAILURE;
```

```
        }

      if (Vector_BS_Mean == NULL)
        {
          Vector_BS_Mean = (double *)malloc(BS_Dimension * sizeof(double));
          if (Vector_BS_Mean == NULL)
            return MEMORY_ALLOCATION_FAILURE;
        }

      if (Inv_Sqrt_BS_Dispersion == NULL)
        {
          Inv_Sqrt_BS_Dispersion = pnl_mat_create(BS_Dimension, BS_Dimension);
          if (Inv_Sqrt_BS_Dispersion == NULL)
            {
              return MEMORY_ALLOCATION_FAILURE;
            }
        }

      if (AuxDBS == NULL)
        {
          AuxDBS = (double *)malloc(BS_Dimension * BS_Dimension * sizeof(double)
          if (AuxDBS == NULL)
            return MEMORY_ALLOCATION_FAILURE;
          for (i = 0; i < BS_Dimension; i++)
            {
              for (j = 0; j < BS_Dimension; j++)
                {
                  aux = 0;
                  for (k = 0; k < BS_Dimension; k++)
                    aux += pnl_mat_get(Sigma, i, k) * pnl_mat_get(Sigma, j, k);
                  AuxDBS[i * BS_Dimension + j] = aux * 0.5;
                }
            }
        }
    }

  return OK;
}

void End_BS()
{
```

```
  if (Sigma != NULL)
    {
      pnl_mat_free(&Sigma);
    }
  if (VDSC != NULL)
    {
      free(VDSC);
      VDSC = NULL;
    }
  if (AuxBS != NULL)
    {
      free(AuxBS);
      AuxBS = NULL;
    }
  if (Vector_BS_Mean != NULL)
    {
      free(Vector_BS_Mean);
      Vector_BS_Mean = NULL;
    }
  if (Inv_Sqrt_BS_Dispersion != NULL)
    {
      pnl_mat_free(&Inv_Sqrt_BS_Dispersion);
    }
  if (AuxDBS != NULL)
    {
      free(AuxDBS);
      AuxDBS = NULL;
    }
  if (Aux_Stock != NULL)
    {
      free(Aux_Stock);
      Aux_Stock = NULL;
    }
}

void Init_Brownian_Bridge(double *Brownian_Bridge, long MonteCarlo_Iterations,
                          int BS_Dimension, double OP_Maturity, int generator)
{
  int i;
  long j;
  double Sqrt_Maturity;
```

```
  /*brownian bridge initialization at the maturity*/
  Sqrt_Maturity = sqrt(OP_Maturity);
  for (j = 0; j < MonteCarlo_Iterations; j++)
    for (i = 0; i < BS_Dimension; i++)
      Brownian_Bridge[j * BS_Dimension + i] = Sqrt_Maturity * pnl_rand_normal(ge
}

void Compute_Brownian_Bridge(double *Brownian_Bridge, double Time, double Step,
                             int BS_Dimension, long MonteCarlo_Iterations,
                             int generator)
{
  double aux1, aux2, *ad, *admax;
  /*backward computation of the brownian bridge at time "Time", knowing its valu
  aux1 = Time / (Time + Step);
  aux2 = sqrt(aux1 * Step);
  ad = Brownian_Bridge;
  admax = Brownian_Bridge + BS_Dimension * MonteCarlo_Iterations;

  for (ad = Brownian_Bridge; ad < admax; ad++)
    *ad = aux1 * (*ad) + aux2 * pnl_rand_normal(generator);
}

void Init_Brownian_Bridge_A(double *Brownian_Bridge, long MonteCarlo_Iterations,
                            int BS_Dimension, double OP_Maturity,
                            int generator)
{
  int i;
  long j;
  double Sqrt_Maturity, aux;
  /*brownian bridge initialization at the maturity with antithetic values*/
  Sqrt_Maturity = sqrt(OP_Maturity);
  for (j = 0; j < MonteCarlo_Iterations / 2; j++)
    {
      for (i = 0; i < BS_Dimension; i++)
        {
          aux = pnl_rand_normal(generator);
          Brownian_Bridge[2 * j * BS_Dimension + i] = Sqrt_Maturity * aux;
          Brownian_Bridge[(2 * j + 1)*BS_Dimension + i] = -Sqrt_Maturity * aux;
        }
    }
```

```
  if (PNL_IS_ODD(MonteCarlo_Iterations))
    {
      for (i = 0; i < BS_Dimension; i++)
        {
          Brownian_Bridge[(MonteCarlo_Iterations - 1)*BS_Dimension + i] = Sqrt_M
        }
    }
}

void Compute_Brownian_Bridge_A(double *Brownian_Bridge, double Time, double Step
                               int BS_Dimension, long MonteCarlo_Iterations,
                               int generator)
{
  int i;
  long n;
  double aux, aux1, aux2;

  /*backward computation of the brownian bridge at time "Time", knowing its valu
  aux1 = Time / (Time + Step);
  aux2 = sqrt(aux1 * Step);
  for (n = 0; n < MonteCarlo_Iterations / 2; n++)
    {
      for (i = 0; i < BS_Dimension; i++)
        {
          aux = pnl_rand_normal(generator);
          Brownian_Bridge[2 * n * BS_Dimension + i] = aux1 * Brownian_Bridge[2 *
          Brownian_Bridge[(2 * n + 1)*BS_Dimension + i] = aux1 * Brownian_Bridge
        }
    }
  if (PNL_IS_ODD(MonteCarlo_Iterations))
    {
      for (i = 0; i < BS_Dimension; i++)
        {
          Brownian_Bridge[(MonteCarlo_Iterations - 1)*BS_Dimension + i] = aux1 *
        }
    }
}

void Backward_Path(double *Paths, double *Brownian_Bridge,
                   double *BS_Spot,
                   double Time,
```

```
                   long MonteCarlo_Iterations, int BS_Dimension)
{
  int j, k;
  long n, auxad;
  double aux;

  /*computation of the BlackScholes paths at time "Time" related to the brownian
  auxad = 0;
  for (n = 0; n < MonteCarlo_Iterations; n++)
    {
      for (j = 0; j < BS_Dimension; j++)
        {
          aux = 0.;
          for (k = 0; k <= j; k++)
            aux += pnl_mat_get(Sigma, j, k) * Brownian_Bridge[auxad + k];
          aux -= Time * AuxBS[j];
          Paths[auxad + j] = BS_Spot[j] * exp(aux);
        }
      auxad += BS_Dimension;
    }
}


void BS_Forward_Path(double *Paths, double *Brownian_Paths, double *BS_Spot, dou
                     long MonteCarlo_Iterations, int BS_Dimension)
{
  int j, k;
  long n;
  double aux, aux1;

  /*computation of the BlackScholes paths at time "Time"*/
  for (n = 0; n < MonteCarlo_Iterations; n++)
    {
      for (j = 0; j < BS_Dimension; j++)
        {
          aux = 0.;
          aux1 = Brownian_Paths[n * BS_Dimension + j];
          for (k = 0; k <= j; k++)
            aux += MGET(Sigma, j, k) * aux1;
          aux -= Time * AuxBS[j];
          Paths[n * BS_Dimension + j] = BS_Spot[j] * exp(aux);
```

```
    }
  }
}


double European_call_price_average(PnlVect *BS_Spot, double Time, double OP_Matu
                                   int BS_Dimension, double BS_Interest_Rate,
                                   PnlVect *BS_Dividend_Rate)
{
  int i;
  double mean, d1, d2, Spot;
  mean = 0.;

  for (i = 0; i < BS_Dimension; i++)
    {
      Spot = GET(BS_Spot, i);
      d1 = log(Spot / (double)Strike) - AuxBS[i] * (OP_Maturity - Time);
      d1 /= sqrt(2 * VDSC[i] * (OP_Maturity - Time));
      d2 = d1 - sqrt(2 * VDSC[i] * (OP_Maturity - Time));
      mean += Spot * exp(-GET(BS_Dividend_Rate, i) * (OP_Maturity - Time)) * cdf
    }
  mean /= (double)BS_Dimension;
  return (mean);
}

double European_call_put_geometric_mean(PnlVect *BS_Spot, double Time, double OP
                                        double Strike, int BS_Dimension, double
                                        PnlVect *BS_Dividend_Rate, double *BS_Vo
                                        double *BS_Correlation, int iscall)
{
  int i, j;
  double r = 0., sig = 0., div = 0., sumsig2 = 0., price, d1, d2, Spot = 1.;
  double S = 0.;

  Spot = pnl_vect_prod(BS_Spot);
  div = pnl_vect_sum(BS_Dividend_Rate);
  for (i = 0; i < BS_Dimension; i++)
    {
      sumsig2 += BS_Volatility[i] * BS_Volatility[i];
    }
```

```
  Spot = POW(Spot, 1. / (double)BS_Dimension);
  r = BS_Interest_Rate - div / (double)BS_Dimension - sumsig2 / (double)(2 * BS_

  for (i = 0; i < BS_Dimension; i++)
    {
      S = 0.;
      for (j = 0; j < BS_Dimension; j++)
        {
          S += BS_Volatility[j] * BS_Correlation[i * BS_Dimension + j];
        }
      sig += S * BS_Volatility[i];
    }

  sig = sqrt(sig) / (double)(BS_Dimension);

  d1 = log(Spot / (double)Strike) + (r + sig * sig / 2.) * (OP_Maturity - Time);
  d1 /= sig * sqrt(OP_Maturity - Time);
  d2 = d1 - sig * sqrt(OP_Maturity - Time);

  if (iscall == TRUE)
    {
      price = Spot * exp((r - BS_Interest_Rate) * (OP_Maturity - Time)) * cdf_no
    }
  else
    {
      price = -Spot * exp((r - BS_Interest_Rate) * (OP_Maturity - Time)) * cdf_n
    }
  return (price);
}

void Compute_Brownian_Paths(double *Brownian_Paths, double Sqrt_Time,
                            int BS_Dimension, long MonteCarlo_Iterations,
                            int generator)
{
  int j;
  int n;
  /*computation of the BlackScholes paths at time "Time"*/
  for (n = 0; n < MonteCarlo_Iterations; n++)
    {
      for (j = 0; j < BS_Dimension; j++)
        {
```

```
          Brownian_Paths[n * BS_Dimension + j] = Sqrt_Time * pnl_rand_normal(gen
        }
    }
}

void Compute_Brownian_Paths_A(double *Brownian_Paths, double Sqrt_Time,
                              int BS_Dimension, long MonteCarlo_Iterations,
                              int generator)
{
  int j, n;
  double aux;
  /*computation of the BlackScholes paths at time "Time"*/
  for (n = 0; n < MonteCarlo_Iterations / 2; n++)
    {
      for (j = 0; j < BS_Dimension; j++)
        {
          aux = Sqrt_Time * pnl_rand_normal(generator);
          Brownian_Paths[2 * n * BS_Dimension + j] = aux;
          Brownian_Paths[(2 * n + 1)*BS_Dimension + j] = -aux;
        }
    }
  if (PNL_IS_ODD(MonteCarlo_Iterations))
    {
      for (j = 0; j < BS_Dimension; j++)
        {
          Brownian_Paths[(MonteCarlo_Iterations - 1)*BS_Dimension + j] = Sqrt_Ti
        }
    }
}


static double BS_Mean(int i, double t, const PnlVect *BS_Spot, double BS_Interes
                      const PnlVect *BS_Dividend_Rate)
{
  /*mean of the ith BlackScholes stock at time t*/
  return pnl_vect_get(BS_Spot, i) * exp(-t * (pnl_vect_get(BS_Dividend_Rate, i)
}

static double BS_Dispersion(int i, int j, double t, int BS_Dimension, const PnlV
                            double BS_Interest_Rate, const PnlVect *BS_Dividend_
{
```

```
  /*coefficient (i,j) of the BlackScholes dispersion matrix at time t*/
  return pnl_vect_get(BS_Spot, i) * pnl_vect_get(BS_Spot, j) *
         exp(-t * (pnl_vect_get(BS_Dividend_Rate, i) + pnl_vect_get(BS_Dividend_
}

void Compute_Inv_Sqrt_BS_Dispersion(double time, int BS_Dimension, const PnlVect
                                    double BS_Interest_Rate, const PnlVect *BS_D
{
  int j, k;
  PnlMat *inv = pnl_mat_create(0, 0);

  /*computation of the inverse of the square root of the BlackScholes dispersion
  for (k = 0; k < BS_Dimension; k++)
    {
      Vector_BS_Mean[k] = BS_Mean(k, time, BS_Spot, BS_Interest_Rate, BS_Dividen
    }
  for (j = 0; j < BS_Dimension; j++)
    {
      for (k = j; k < BS_Dimension; k++)
        {
          pnl_mat_set(Inv_Sqrt_BS_Dispersion,
                      j, k, BS_Dispersion(j, k, time, BS_Dimension, BS_Spot,
                                          BS_Interest_Rate, BS_Dividend_Rate));
        }
    }

  pnl_mat_chol(Inv_Sqrt_BS_Dispersion);
  pnl_mat_lower_inverse(inv, Inv_Sqrt_BS_Dispersion);
  pnl_mat_clone(Inv_Sqrt_BS_Dispersion, inv);
  pnl_mat_free(&inv);
}

void NormalisedPaths(double *Paths, double *PathsN, long MonteCarlo_Iterations,
                     int BS_Dimension)
{
  long i;
  int j, k;
  double aux;

  /*BlackScholes paths normalization (mean 0, variance Id).*/
  for (i = 0; i < MonteCarlo_Iterations; i++)
```

```
    {
      for (k = 0; k < BS_Dimension; k++)
        {
          PathsN[i * BS_Dimension + k] = Paths[i * BS_Dimension + k] - Vector_BS
        }
    }
  for (i = 0; i < MonteCarlo_Iterations; i++)
    {
      for (j = 0; j < BS_Dimension; j++)
        {
          aux = 0;
          for (k = 0; k <= j; k++)
            {
              aux += pnl_mat_get(Inv_Sqrt_BS_Dispersion, j, k) * PathsN[i * BS_D
            }
          Aux_Stock[j] = aux;
        }
      for (j = 0; j < BS_Dimension; j++)
        PathsN[i * BS_Dimension + j] = Aux_Stock[j];
    }
}

void ForwardPath(double *Path, double *Initial_Stock, int Initial_Time,
                 int Number_Dates, int BS_Dimension, double Step, double Sqrt_St
                 int generator)
{
  int i, j, k;
  double aux;
  double *SigmapjmBS_Dimensionpk;

  /*computation of a BlackScholes path between times "Initial_Time" and "Initial
  for (j = 0; j < BS_Dimension; j++) Path[Initial_Time * BS_Dimension + j] = Ini

  for (i = Initial_Time + 1; i < Initial_Time + Number_Dates; i++)
    {
      for (j = 0; j < BS_Dimension; j++)
        {
          Aux_Stock[j] = Sqrt_Step * pnl_rand_normal(generator);
        }
      SigmapjmBS_Dimensionpk = pnl_mat_lget(Sigma, 0, 0);
      for (j = 0; j < BS_Dimension; j++)
```

```
        {
          aux = 0.;
          for (k = 0; k <= j; k++)
            {
              aux += (*SigmapjmBS_Dimensionpk) * Aux_Stock[k];
              SigmapjmBS_Dimensionpk++;
            }
          SigmapjmBS_Dimensionpk += BS_Dimension - j - 1;
          aux -= Step * AuxBS[j];
          Path[i * BS_Dimension + j] = Path[(i - 1) * BS_Dimension + j] * exp(au
        }
    }
}

void BlackScholes_Transformation(double Instant, double *BS, double *B, int BS_D
{
  double aux;
  int j, k;
  /*computation a the BlackScholes stock related to the brownian motion value B*
  for (j = 0; j < BS_Dimension; j++)
    {
      aux = 0.;
      for (k = 0; k <= j; k++)
        aux += pnl_mat_get(Sigma, j, k) * B[k];
      aux -= Instant * AuxBS[j];
      BS[j] = BS_Spot[j] * exp(aux);
    }
}

double Discount(double Time, double BS_Interest_Rate)
{
  /*discounting factor*/
  return exp(-BS_Interest_Rate * Time);
}

int BS_Transition_Allocation(int BS_Dimension, double Step)
{
  int i;
  /*memory allocation of the variables needed by the function BS_TD*/
  Aux_BS_TD_1 = (double *)malloc(BS_Dimension * sizeof(double));
  if (Aux_BS_TD_1 == 0) return MEMORY_ALLOCATION_FAILURE;
```

```
  Aux_BS_TD_2 = (double *)malloc(BS_Dimension * sizeof(double));
  if (Aux_BS_TD_2 == 0)return MEMORY_ALLOCATION_FAILURE;

  InvSigma = pnl_mat_create(0, 0);
  if (InvSigma == 0)return MEMORY_ALLOCATION_FAILURE;

  /* Sigma is a Cholesky factorisation */
  pnl_mat_lower_inverse(InvSigma, Sigma);

  /* determinant of InvSigma */
  DetInvSigma = 1;
  for (i = 0; i < BS_Dimension; i++)
    DetInvSigma *= pnl_mat_get(InvSigma, i , i);

  /*normalization constant of the density */
  Norm_BS_TD = exp(-BS_Dimension * 0.5 * log(2.*M_PI * Step));
  return OK;
}

void BS_Transition_Liberation(char *ErrorMessage, int BS_Dimension, double Step)
{
  if (Aux_BS_TD_1 != NULL)
    {
      free(Aux_BS_TD_1);
      Aux_BS_TD_1 = NULL;
    }
  if (Aux_BS_TD_2 != NULL)
    {
      free(Aux_BS_TD_2);
      Aux_BS_TD_2 = NULL;
    }
  if (InvSigma != NULL) pnl_mat_free(&InvSigma);
}

double BS_TD(double *X, double *Z, int BS_Dimension, double Step)
{
  int i, j;
  double aux1, aux2;

  /* density function of the BlackScholes stock transition kernel at time "Step"
```

```
  for (i = 0; i < BS_Dimension; i++)
    {
      Aux_BS_TD_1[i] = log(Z[i] / X[i]) + Step * AuxBS[i];
    }
  aux1 = Z[0];
  for (i = 1; i < BS_Dimension; i++)
    {
      aux1 *= Z[i];
    }
  if (aux1 == 0)
    {
      return -1;
    }
  else
    {
      for (i = 0; i < BS_Dimension; i++)
        {
          Aux_BS_TD_2[i] = 0;
          for (j = 0; j <= i; j++)
            {
              Aux_BS_TD_2[i] += pnl_mat_get(InvSigma, i , j) * Aux_BS_TD_1[j];
            }
        }
      aux2 = 0;
      for (i = 0; i < BS_Dimension; i++)
        {
          aux2 += Aux_BS_TD_2[i] * Aux_BS_TD_2[i];
        }
      aux2 = exp(-aux2 / (2.*Step));
      return Norm_BS_TD * DetInvSigma * aux2 / aux1;
    }
}

void BS_Forward_Step(double *Stock, double *Initial_Stock, int BS_Dimension,
                     double Step, double Sqrt_Step,
                     int generator)
{
  int j, k;
  double Aux;

  /*BlackScholes stock knowing "Initial_Stock" at the preceding time*/
```

```
  for (j = 0; j < BS_Dimension; j++)
    Aux_Stock[j] = Sqrt_Step * pnl_rand_normal(generator);

  for (j = 0; j < BS_Dimension; j++)
    {
      Aux = 0.;
      for (k = 0; k <= j; k++)
        Aux += pnl_mat_get(Sigma, j , k) * Aux_Stock[k];

      Aux -= Step * AuxBS[j];
      Stock[j] = Initial_Stock[j] * exp(Aux);
    }
}
/*******************************************************/
/*        Routines for LS importance sampling      */
/*******************************************************/
/*array copy*/
void Xcopy(double *Original, double *Copy, int dim)
{
  int i;
  for (i = 0; i < dim; i++)
    {
      Copy[i] = Original[i];
    }
  return;
}

/*initialize RM drift*/
void InitThetasigma(double *theta, double *thetasigma, int BS_Dimension)
{
  int i, j;
  for (i = 0; i < BS_Dimension; i++)
    {
      thetasigma[i] = 0.0;
      for (j = 0; j <= i; j++) thetasigma[i] += pnl_mat_get(Sigma, i , j) * thet
    }

  return;
}
/*build drifted path from undrifted*/
void ThetaDriftedPaths(double *Paths, double *thetasigma, double Time, long AL_M
```

```
{
  long i;
  int j;
  for (i = 0; i < AL_MonteCarlo_Iterations; i++)
    {
      for (j = 0; j < BS_Dimension; j++)
        {
          Paths[i * BS_Dimension + j] *= exp(thetasigma[j] * Time);
        }
    }
}
/*RM step*/
void RMsigma(double *sigma, int BS_Dimension)
{
  int j, i;
  for (i = 0; i < BS_Dimension; i++)
    {
      for (j = 0; j < BS_Dimension; j++)
        sigma[i * BS_Dimension + j] = pnl_mat_get(Sigma, i , j);
    }
  return;
}
/*Stocking of gaussian array for the Robbins-Monro routine*/
void gauss_stock(double *normalvect, int N, int generator)
{
  int l;
  for (l = 0; l < N; l++) normalvect[l] = pnl_rand_normal(generator);
}
```