Help

```
/* Longstaff & Schwartz algorithm, backward simulated brow
    nian paths  */
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

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

static double *FP=NULL, *Paths=NULL, *PathsN=NULL;
static double *Brownian_Bridge=NULL;
static PnlMat *M = NULL;
static PnlVect *AuxR=NULL, *Res=NULL, *VBase=NULL;

/* only for importance sampling*/
static double *theta=NULL,*thetasigma=NULL;
static PnlBasis *Basis = NULL;

static int LoScB_Allocation(long AL_MonteCarlo_Iterations,
                            int AL_Basis_Dimension, int BS_
    Dimension)
{
    if (FP==NULL) FP=(double*)malloc(AL_MonteCarlo_Iteratio
    ns*sizeof(double));
    if (FP==NULL)  return MEMORY_ALLOCATION_FAILURE;

    if (Paths==NULL) Paths=(double*)malloc(AL_MonteCarlo_
    Iterations*BS_Dimension*sizeof(double));
    if (Paths==NULL)  return MEMORY_ALLOCATION_FAILURE;

    /* only usefull for normalised L&S, suboptimal but ...
    */
    if (PathsN==NULL){
        PathsN=(double*)malloc(AL_MonteCarlo_Iterations*BS_
```

```
    Dimension*sizeof(double));
    }
    if (PathsN==NULL)  return MEMORY_ALLOCATION_FAILURE;

    if (M==NULL) M=pnl_mat_create(AL_Basis_Dimension, AL_
    Basis_Dimension);
    if (M==NULL) return MEMORY_ALLOCATION_FAILURE;

    if (Brownian_Bridge==NULL){
        Brownian_Bridge=(double*)malloc(AL_MonteCarlo_Itera
    tions*BS_Dimension*sizeof(double));
    }
    if (Brownian_Bridge==NULL) return MEMORY_ALLOCATION_FAI
    LURE;

    if (Res==NULL) Res=pnl_vect_create (AL_Basis_Dimension)
    ;
    if (Res==NULL) return MEMORY_ALLOCATION_FAILURE;

    if (AuxR==NULL) AuxR = pnl_vect_create (AL_Basis_Dimens
    ion);
    if (AuxR==NULL) return MEMORY_ALLOCATION_FAILURE;

    if (VBase==NULL) VBase = pnl_vect_create (AL_Basis_Dim
    ension);
    if (VBase==NULL) return MEMORY_ALLOCATION_FAILURE;

    return OK;
}

static int Theta_Allocation (int BS_Dimension)
{
    if (theta==NULL)  theta=(double*)malloc(BS_Dimension*si
    zeof(double));
    if (theta==NULL) return MEMORY_ALLOCATION_FAILURE;
    if (thetasigma==NULL) thetasigma=(double*)malloc(BS_Dim
    ension*sizeof(double));
    if (thetasigma==NULL) return MEMORY_ALLOCATION_FAILURE;
    return OK;
}
```

```
static void LoScB_Liberation()
{
    if (FP!=NULL) {free(FP); FP=NULL;}
    if (Brownian_Bridge!=NULL) {free(Brownian_Bridge); Brow
    nian_Bridge=NULL;}
    if (Paths!=NULL) {free(Paths); Paths=NULL;}
    if (PathsN!=NULL) {free(PathsN); PathsN=NULL;}
    if (M!=NULL) {pnl_mat_free (&M);}
    if (Res!=NULL) {pnl_vect_free (&Res); }
    if (AuxR!=NULL) {pnl_vect_free (&AuxR);}
    if (VBase!=NULL) {pnl_vect_free (&VBase);}
}


static void Theta_Liberation()
{
    if (thetasigma!=NULL){ free(thetasigma); thetasigma=NUL
    L;}
    if (theta!=NULL) {free(theta); theta=NULL;}
}

/*Compute the Girsanov Martingale (change of measure)*/
static double Girsanovfactor(double *BrownianPath,double
    Time,int BS_Dimension)
{
    int jj;
    double auxgirs;

    auxgirs=0.0;
    /*---L(theta,t)=exp[-theta*W(t)-0.5*thetaš*t]----*/
    for(jj=0;jj<BS_Dimension;jj++)
        auxgirs+=(-0.5*theta[jj]*theta[jj]*Time-theta[jj]*
    BrownianPath[jj]);
    return(exp(auxgirs));
}

/*Creal sequence for Robbins Monro convergence*/
static double gamma_RM(int n,double a,double b)
{
    return (a/(b+n));
```

```
}

static double rmstep(int n)
{
    return (sqrt(log(n+1)/10.0)+100.);
}


/*Robbins Monro algorithm, adaptation of code used by Arou
    na,since Premia 4*/
static int rmgraphic(NumFunc_nd *p, PnlVect *Stock,double
    t,double r,
                     double *divid,int BS_Dimension,double
    *teta,
                     int generator)
{
    int RM=500000;/*NUMBER OF RM iteration*/
    int i=0,j=0,ii=0;
    int sig_itere=0;
    double expo=0,S_i;
    double sqrt_T=sqrt(t);
    double x_1=0.0925,x_2=0.0725;
    double *m_Theta=NULL,*Vol_tp=NULL,*vol=NULL,*Normalvec
    t=NULL,*m_Mu=NULL;
    double *m_UnderlyingAsset=NULL,*m_sigma=NULL;
    double vol_T,a,b,dot1,dot2,payoff,payoffcarre,val_test,
    temp;

    PnlVect Vm_UnderlyingAsset;
    Vm_UnderlyingAsset.size=BS_Dimension;
    /*memory allocation*/
    if (Normalvect==NULL) Normalvect=(double*)malloc(BS_Dim
    ension*RM*sizeof(double));
    if (Normalvect==NULL) return MEMORY_ALLOCATION_FAILURE;
    if (m_Mu==NULL)  m_Mu=(double*)malloc(BS_Dimension*size
    of(double));
    if (m_Mu==NULL) return MEMORY_ALLOCATION_FAILURE;
    if (m_sigma==NULL) m_sigma=(double*)malloc(BS_Dimensio
    n*BS_Dimension*sizeof(double));
    if (m_sigma==NULL) return MEMORY_ALLOCATION_FAILURE;
    if (m_UnderlyingAsset==NULL) m_UnderlyingAsset=(double*
```

```
)malloc(BS_Dimension*sizeof(double));
if (m_UnderlyingAsset==NULL) return MEMORY_ALLOCATION_
FAILURE;
if (vol==NULL) vol=(double *)malloc(sizeof(double)*(BS_
Dimension+1));
if (vol==NULL) return MEMORY_ALLOCATION_FAILURE;
if (Vol_tp==NULL) Vol_tp=(double *)malloc(sizeof(
double)*(BS_Dimension+1));
if (Vol_tp==NULL) return MEMORY_ALLOCATION_FAILURE;
if (m_Theta==NULL) m_Theta=(double *)malloc(sizeof(
double)*(BS_Dimension+1));
if (m_Theta==NULL) return MEMORY_ALLOCATION_FAILURE;

/*initialization of gaussian variables*/
gauss_stock(Normalvect,BS_Dimension*RM, generator);

/*initialisation of the drift m_Mu and of the
volatility */
for(i=0;i<BS_Dimension;i++) m_Mu[i]=0;
RMsigma(m_sigma,BS_Dimension);

for(i=0;i<BS_Dimension;i++){
    vol[i]=0.0;
    for(j=0;j<=i;j++){
        vol[i]+=(m_sigma[i*BS_Dimension+j]*m_sigma[i*
BS_Dimension+j]);
    }

}
/*a is the step of R-M iterations: very important to ha
ve fast*/
/*convergence to sub-optimal theta*/
a=0.19;
b=1.0;
for(ii=0;ii<RM;ii++){
    for(i=0;i<BS_Dimension;i++)
    {
        vol_T=0;
        for(j=0;j<BS_Dimension;j++)
            vol_T+=Normalvect[j+ii*BS_Dimension]*m_si
gma[i*BS_Dimension+j];
```

```
        Vol_tp[i]=vol_T;
    }
    dot1=0.0;dot2=0.0;
    for(i=0;i<BS_Dimension;i++)
    {
        S_i=(r-divid[i]-0.5*vol[i])*t+Vol_tp[i]*sqrt_T;
        m_UnderlyingAsset[i]=exp(S_i)*Stock->array[i];
        dot1+=Normalvect[i+ii*BS_Dimension]*m_Mu[i];
        dot2+=m_Mu[i]*m_Mu[i];
    }
    Vm_UnderlyingAsset.array = m_UnderlyingAsset;
    payoff=exp(-r*t)*p->Compute(p->Par, &Vm_Underlying
Asset);
    payoffcarre=payoff*payoff;
    expo=exp(-dot1+0.5*dot2);
    val_test=0;
    for(i=0;i<BS_Dimension;i++)
    {
        temp=(m_Mu[i]-Normalvect[i+ii*BS_Dimension])*
expo*payoffcarre;
        m_Theta[i]=temp;
        val_test+=pow(m_Mu[i]-gamma_RM(ii,a,b)*temp,2);
    }
    val_test=sqrt(val_test);
    if(val_test<=rmstep(sig_itere))
    {
        for(i=0;i<BS_Dimension;i++)
            m_Mu[i]=m_Mu[i]-gamma_RM(ii,a,b)*m_Theta[i]
;
    }
    else
    {
        if(sig_itere-2*(sig_itere/2)==0)
            for(i=0;i<BS_Dimension;i++)
                m_Mu[i]=x_1;
        else
            for(i=0;i<BS_Dimension;i++)
                m_Mu[i]=x_2;
        sig_itere+=1;
    }
```

```
    }
    for(i=0;i<BS_Dimension;i++) {teta[i]=m_Mu[i];}
    /*memory liberation*/
    if (Normalvect!=NULL) {free(Normalvect); Normalvect=NUL
    L; }
    if (m_UnderlyingAsset!=NULL) { free(m_UnderlyingAsset);
     m_UnderlyingAsset=NULL; }
    if (m_Mu!=NULL) {free(m_Mu); m_Mu=NULL;}
    if (vol!=NULL) {free(vol); vol=NULL;}
    if (Vol_tp!=NULL) {free(Vol_tp); Vol_tp=NULL;}
    if (m_Theta!=NULL) {free(m_Theta); m_Theta=NULL; }

    return OK;
}


static void Regression( long AL_MonteCarlo_Iterations,
    NumFunc_nd *p,
                    int AL_Basis_Dimension, int BS_Dim
    ension, int Time,
                    int AL_PayOff_As_Regressor, int us
    e_normalised_regressor,
                    int use_importance_sampling,
    double step)
{
    int i,j,k;
    double AuxOption, tmp;
    double *PathspkmDimBS=Paths, *PathsNpkmDimBS=Paths;
    PnlVect VStock;
    long InTheMonney=0;
    VStock.size=BS_Dimension;

    if(use_normalised_regressor)
        PathsNpkmDimBS=PathsN;

    pnl_vect_set_double (AuxR, 0.0);
    pnl_mat_set_double (M, 0.0);

    for(k=0;k<AL_MonteCarlo_Iterations;k++)
    {
        /*kth regressor value*/
```

```
    VStock.array=&(PathspkmDimBS[k*BS_Dimension]);
    AuxOption=p->Compute(p->Par, &VStock);
    if (use_importance_sampling)
        AuxOption*=Girsanovfactor(Brownian_Bridge+k*BS_
Dimension,(double)Time*step,BS_Dimension);
    /*only the at-the-monney path are taken into accoun
t*/
    if (AuxOption>0)
    {
        InTheMonney++;
        /*value of the regressor basis on the kth path*
/
        if (AL_PayOff_As_Regressor==1)
        {
            /*here, the payoff function is introduced
in the regression basis*/
            pnl_vect_set (VBase, 0, AuxOption);
            for (i=1;i<AL_Basis_Dimension;i++){
                pnl_vect_set (VBase, i, pnl_basis_i(
Basis,&(PathsNpkmDimBS[k*BS_Dimension]),i-1));
            }
        }
        else
        {
            for (i=0;i<AL_Basis_Dimension;i++){
                pnl_vect_set (VBase, i, pnl_basis_i(
Basis,&(PathsNpkmDimBS[k*BS_Dimension]),i));
            }
        }
        /*empirical regressor dispersion matrix*/
        for (i=0;i<AL_Basis_Dimension;i++)
            for (j=0;j<AL_Basis_Dimension;j++)
            {
                tmp = pnl_mat_get (M, i, j);
                pnl_mat_set (M, i, j , tmp + pnl_vect_
get (VBase, i) * pnl_vect_get (VBase,j));
            }
        /*auxiliary for regression formulae*/
        for (i=0;i<AL_Basis_Dimension;i++){
            tmp = pnl_vect_get(AuxR, i);
            pnl_vect_set (AuxR, i, FP[k] * pnl_vect_get
```

```
     (VBase,i) + tmp);
            }
        }
    }
    if (InTheMonney==0)
    {
        pnl_vect_set_double (Res, 0);
    }
    else
    {
        /*normalisation*/
        pnl_vect_div_double (AuxR, InTheMonney);
        pnl_mat_div_double (M, InTheMonney);
        pnl_vect_clone (Res, AuxR);
        /* solve in the least square sense, using a QR de
    composition */
        pnl_mat_ls (M, Res);
    }
}

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

/*see the documentation for the parameters meaning*/
int LoScB(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 name_basis,
          int AL_Basis_Dimension,
          int OP_Exercice_Dates,
          int AL_PayOff_As_Regressor,
          int AL_Antithetic,
```

```
        int use_normalised_regressor,
        int use_importance_sampling,
        double *AL_FPrice)
{
    double AuxOption,AuxScal,DiscountStep,Step;
    long i;
    int k,l, init_mc, init;
    int BS_Dimension = BS_Spot->size;
    double *paths; /* = Paths but changed to PathsN, when
    use_normalised_regressor*/
    PnlVect VStock;
    *AL_FPrice=0.;

    /* 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;

    if (use_importance_sampling)
    {
        /*memory allocation for the importance sampling (dr
    ift) variables*/
        init=Theta_Allocation(BS_Dimension);
        if (init!=OK) return init;
    }
    /* initialisation of BS */
    init=Init_BS(BS_Dimension, BS_Volatility->array, BS_
    Correlation,
                 BS_Interest_Rate, BS_Dividend_Rate->array)
    ;
    if (init!=OK) return init;

    /*Initialization of the regression basis*/
    Basis = pnl_basis_create (name_basis, AL_Basis_Dimensio
    n, BS_Dimension);

    /*time step*/
    Step=OP_Maturity/(double)(OP_Exercice_Dates-1);
    /*discounting factor for a time step*/
```

```
DiscountStep=exp(-BS_Interest_Rate*Step);

if (use_importance_sampling)
{
    /*initialization of drift "theta" via a Robbins-
Monro Algorithm for the european option*/
    init=rmgraphic(p, BS_Spot,OP_Maturity,BS_Interest_
Rate,BS_Dividend_Rate->array,
                   BS_Dimension,theta, generator);
    if (init!=OK) return init;
    /*The theta given by RM driftes the gaussian variable
     *who is used to compute
     *W(OP_Maturity)={sqrt(OP_Maturity)*gaussian. As a
     *consequence, the theta that  has to be used for
W is
     *given as: thetaUSA=thetaRM/sqrt(OP_Maturity)*/
    for(l=0;l<BS_Dimension;l++) theta[l]/=sqrt(OP_Matu
rity);

    /*Initialization of extra-drift */
    InitThetasigma(theta,thetasigma,BS_Dimension);
}

/*memory allocation of the algorithm's variables*/
init=LoScB_Allocation(AL_MonteCarlo_Iterations,AL_Basi
s_Dimension, BS_Dimension);
if (init!=OK) return init;
paths=Paths;

if (AL_Antithetic)
    /*here, the brownian bridge is initialised with an
tithetic paths*/
    Init_Brownian_Bridge_A(Brownian_Bridge,AL_
MonteCarlo_Iterations,
                           BS_Dimension,OP_Maturity,     generator);
else
    Init_Brownian_Bridge(Brownian_Bridge,AL_MonteCarlo_
Iterations,
                         BS_Dimension,OP_Maturity,     generator);

/*computation of the BlackScholes paths at the maturit
```

```
y related to Brownian_Bridge*/
Backward_Path(Paths,Brownian_Bridge,BS_Spot->array,OP_
Maturity,
              AL_MonteCarlo_Iterations,BS_Dimension);

if (use_importance_sampling)
    /*adjusting drift for the BlackScholes paths at th
e maturity*/
    ThetaDriftedPaths(Paths,thetasigma,OP_Maturity,AL_
MonteCarlo_Iterations,BS_Dimension);

/*initialisation of the payoff values at the maturity*/
for (i=0;i<AL_MonteCarlo_Iterations;i++)
{
    VStock.size=BS_Dimension;
    VStock.array = &(Paths[i*BS_Dimension]);
    FP[i]=p->Compute(p->Par, &VStock);
    if (use_importance_sampling)
        FP[i] *= Girsanovfactor(Brownian_Bridge+i*BS_
Dimension,OP_Maturity,BS_Dimension);
    if (FP[i]>0) FP[i]=DiscountStep*FP[i];
}

for (k=OP_Exercice_Dates-2;k>=1;k--)
{
    if (AL_Antithetic)
        /*here, the brownian bridge is computed with an
tithetic paths*/
        Compute_Brownian_Bridge_A(Brownian_Bridge,k*
Step,Step,BS_Dimension,
                                  AL_MonteCarlo_Itera
tions, generator);
    else
        Compute_Brownian_Bridge(Brownian_Bridge,k*Step,
Step,BS_Dimension,
                                AL_MonteCarlo_Iteratio
ns, generator);


    /*computation of the BlackScholes paths at time k
related to Brownian_Bridge*/
```

```
    Backward_Path(Paths,Brownian_Bridge,BS_Spot->array,
(double)k*Step,
                  AL_MonteCarlo_Iterations,BS_Dimensio
n);


    if (use_normalised_regressor)
    {
        /*computation of the inverse of the BlackScho
les
         * dispersion matrix used in the normalisation
paths
         * procedure*/
        Compute_Inv_Sqrt_BS_Dispersion((double)k*Step,
BS_Dimension,BS_Spot,
                                       BS_Interest_Ra
te,BS_Dividend_Rate);

        /*the regression is done with respect to the
normalised
         * BlackScholes paths (see the documentation)*/
        NormalisedPaths(Paths,PathsN,AL_MonteCarlo_
Iterations,BS_Dimension);
        paths=PathsN;
    }

    if (use_importance_sampling)
        /*adjusting drift for the BlackScholes paths
at time k*/
        ThetaDriftedPaths(Paths,thetasigma,(double)k*
Step,AL_MonteCarlo_Iterations,BS_Dimension);

    /*regression procedure*/
    Regression(AL_MonteCarlo_Iterations,p, AL_Basis_Dim
ension,
               BS_Dimension,k,AL_PayOff_As_Regressor,
use_normalised_regressor,
               use_importance_sampling, Step);
    /* dynamical programming*/
    for (i=0;i<AL_MonteCarlo_Iterations;i++)
    {
```

```
        /*exercise value*/
        VStock.size=BS_Dimension;
        VStock.array = &(Paths[i*BS_Dimension]);
        AuxOption=p->Compute(p->Par, &VStock);
        if (use_importance_sampling)
            AuxOption *= Girsanovfactor(Brownian_Brid
ge+i*BS_Dimension,(double)k*Step,BS_Dimension);
        /*approximated continuation value, only the at-
the-monney paths are taken into account*/
        if (AuxOption>0)
        {
            /* if k is greater than or equal to
             * AL_PayOff_As_Regressor, the payoff
function is
             * introduced to the regression basis*/
            if (AL_PayOff_As_Regressor==1)
            {
                AuxScal=AuxOption*pnl_vect_get (Res, 0)
;
                for (l=1;l<AL_Basis_Dimension;l++)
                    AuxScal+=pnl_basis_i(Basis,paths+i*
BS_Dimension,l-1)*pnl_vect_get (Res, l);
            }
            else
            {
                AuxScal=0.;
                for (l=0;l<AL_Basis_Dimension;l++){
                    AuxScal+=pnl_basis_i(Basis,paths+i*
BS_Dimension,l)*pnl_vect_get (Res,l);
                }
            }
            /* AuxScal contains the approximated conti
nuation value*/
            /* if the continuation value is less than
the exercise value,
             * the optimal stopping time is modified*/
            if (AuxOption>AuxScal)
                FP[i]=AuxOption;

        }
        /*Discount for a time step*/
```

```
          FP[i]*=DiscountStep;
        }
    }
    /*at time 0, the conditionnal expectation reduces to an
     expectation*/
    for (i=0;i<AL_MonteCarlo_Iterations;i++){
        *AL_FPrice+=FP[i];
    }
    *AL_FPrice/=(double)AL_MonteCarlo_Iterations;

    /*output of the algorithm*/
    *AL_FPrice=MAX(p->Compute(p->Par, BS_Spot),*AL_FPrice);

    if (use_importance_sampling)
        Theta_Liberation();
    pnl_basis_free (&Basis);
    End_BS();
    Close();
    return OK;
}



int CALC(MC_LongstaffSchwartzND)(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 = pnl_vect_compact_to_pnl_vect (ptMod->S0.Val.V_PN
    LVECTCOMPACT);
    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, 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_PI
NT; 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;



/* If we use importance sampling : activate
 * payoff_as_regressor and deactivates normalised_reg
ressor */
if (Met->Par[7].Val.V_BOOL)
{
    Met->Par[5].Val.V_BOOL=1; /* payoff as regressor */
    Met->Par[7].Val.V_BOOL=0;  /* normalised regressor
s */
}

res=LoScB(spot,
          ptOpt->PayOff.Val.V_NUMFUNC_ND,
          ptOpt->Maturity.Val.V_DATE-ptMod->T.Val.V_DA
TE,
          r, divid, sig,
          BS_cor,
          Met->Par[0].Val.V_LONG,
          Met->Par[1].Val.V_ENUM.value,
          Met->Par[2].Val.V_ENUM.value,
          Met->Par[3].Val.V_INT,
          Met->Par[4].Val.V_INT,
          Met->Par[5].Val.V_ENUM.value,
          Met->Par[6].Val.V_ENUM.value,
          Met->Par[7].Val.V_ENUM.value,
          Met->Par[8].Val.V_ENUM.value,
```

```
              &(Met->Res[0].Val.V_DOUBLE));
    pnl_vect_free(&divid);
    free(BS_cor);
    pnl_vect_free (&spot);
    pnl_vect_free (&sig);
    return res;
}

static int CHK_OPT(MC_LongstaffSchwartzND)(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->HelpFilenameHint = "mc_longstaffschwatrz_nd";
        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_ENUM.value=0;
        Met->Par[2].Val.V_ENUM.members=&PremiaEnumBasis;
        Met->Par[3].Val.V_INT=9;
        Met->Par[4].Val.V_INT=10;
        Met->Par[5].Val.V_ENUM.value=1;
        Met->Par[5].Val.V_ENUM.members=&PremiaEnumBool;
        Met->Par[6].Val.V_ENUM.value=0;
        Met->Par[6].Val.V_ENUM.members=&PremiaEnumBool;
        Met->Par[7].Val.V_ENUM.value=0;
        Met->Par[7].Val.V_ENUM.members=&PremiaEnumBool;
        Met->Par[8].Val.V_ENUM.value=0;
        Met->Par[8].Val.V_ENUM.members=&PremiaEnumBool;
    }
    return OK;
```

```
}

PricingMethod MET(MC_LongstaffSchwartzND)=
{
    "MC_LongstaffSchwartz_ND",
    {{"N iterations",LONG,{100},ALLOW},
     {"RandomGenerator",ENUM,{0},ALLOW},
     {"Basis",ENUM,{1},ALLOW},
     {"Dimension Approximation",INT,{100},ALLOW},
     {"Number of Exercise Dates",INT,{100},ALLOW},
     {"Use Payoff as Regressor",ENUM,{1},ALLOW},
     {"Use Antithetic Variables",ENUM,{1},ALLOW},
     {"Use Normalised Regressors",ENUM,{0},ALLOW},
     {"Use Importance Sampling",ENUM,{0},ALLOW},
     {" ",PREMIA_NULLTYPE,{0},FORBID}},
    CALC(MC_LongstaffSchwartzND),
    {{"Price",DOUBLE,{100},FORBID},
     {" ",PREMIA_NULLTYPE,{0},FORBID}},
    CHK_OPT(MC_LongstaffSchwartzND),
    CHK_mc,
    MET(Init)
};
```

References