Help

```
/*Tsitsiklis & VanRoy algorithm, backward paths simulation*
  /
#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 "var.h"
#include "pnl/pnl_random.h"
#include "pnl/pnl_matrix.h"

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

static PnlBasis *Basis;

static int TsRoB_Allocation(long AL_MonteCarlo_Iterations,
               int AL_Basis_Dimension,int BS_Dimens
    ion)
{
  if (FP==NULL)
  FP=(double*)malloc(AL_MonteCarlo_Iterations*sizeof(
    double));
  if (FP==NULL) { return MEMORY_ALLOCATION_FAILURE;  }
  if (Paths==NULL)
  Paths=(double*)malloc(AL_MonteCarlo_Iterations*BS_Dimens
    ion*sizeof(double));
  if (Paths==NULL){ return MEMORY_ALLOCATION_FAILURE; }
  if (Vector_Price==NULL)
  Vector_Price=(double*)malloc(AL_MonteCarlo_Iterations*si
    zeof(double));
  if (Vector_Price==NULL) {return MEMORY_ALLOCATION_FAILU
    RE;  }
```

```
  if (M==NULL) M=pnl_mat_create(AL_Basis_Dimension, AL_Basi
    s_Dimension);
  if (M==NULL) return MEMORY_ALLOCATION_FAILURE;

  if (Brownian_Bridge==NULL)
  Brownian_Bridge=(double*)malloc(AL_MonteCarlo_Iterations
    *BS_Dimension*sizeof(double));
  if (Brownian_Bridge==NULL){return MEMORY_ALLOCATION_FAILU
    RE;  }

  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_Dimensio
    n);
  if (AuxR==NULL) return MEMORY_ALLOCATION_FAILURE;

  if (VBase==NULL) VBase = pnl_vect_create (AL_Basis_Dimens
    ion);
  if (VBase==NULL) return MEMORY_ALLOCATION_FAILURE;
  return OK;
}

static void TsRoB_Liberation()
{
  if (FP!=NULL) {free(FP); FP=NULL; }
  if (Brownian_Bridge!=NULL) {  free(Brownian_Bridge);
    Brownian_Bridge=NULL;  }
  if (Paths!=NULL) {  free(Paths);  Paths=NULL;  }
  if (Vector_Price!=NULL) {  free(Vector_Price);Vector_
    Price=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 Compute_Vector_Price(long AL_MonteCarlo_Itera
    tions, NumFunc_nd *p,
                             int OP_Exercise_Dates,
```

```
                int AL_Basis_Dimension, int BS_Dim
    ension, int Time,
                int AL_PayOff_As_Regressor,double
    DiscountStep)
{
  long k;
  int i;
  double Aux,AuxOption,AuxScal;
  PnlVect VStock;
  VStock.size=BS_Dimension;

  for(k=0;k<AL_MonteCarlo_Iterations;k++){
    VStock.array=&(Paths[k*BS_Dimension]);
    AuxOption=DiscountStep*p->Compute(p->Par, &VStock);
    if (Time==OP_Exercise_Dates-1){
    /*initialisation of the payoff values and of the     dynamical programming p
    Vector_Price[k]=AuxOption;
    FP[k]=AuxOption;
  } else {
      /*computation of the regressor values*/
    if (AL_PayOff_As_Regressor<=Time){
    /*here, the payoff function is introduced in the
         * regression basis*/
        VStock.array=&(Paths[k*BS_Dimension]);
    pnl_vect_set (VBase, 0, p->Compute(p->Par, &VStock));
    for (i=1;i<AL_Basis_Dimension;i++){
      pnl_vect_set (VBase, i, pnl_basis_i(Basis,Paths+k*
    BS_Dimension,i-1));
    }
    } else {
    for (i=0;i<AL_Basis_Dimension;i++){
      pnl_vect_set (VBase, i, pnl_basis_i(Basis,Paths+k*
    BS_Dimension,i));
    }
    }

    AuxScal=0;
    for (i=0;i<AL_Basis_Dimension;i++){
    AuxScal += pnl_vect_get (Res, i) * pnl_vect_get (VB
    ase, i);
    }
```

```
    /*dynamical programming for the backward price*/
    AuxScal*=DiscountStep;
    Aux=MAX(AuxOption,AuxScal);
    Vector_Price[k]=Aux;

    /*dynamical programming for the forward price*/
    if (AuxOption==Aux){
    FP[k]=Aux;
    } else {
    FP[k]*=DiscountStep;
    }
  }
  }
}

static void Regression(long AL_MonteCarlo_Iterations,  int
    OP_Exercice_Dates, NumFunc_nd *p,
             int AL_Basis_Dimension, int BS_Dimension,
     int Time,
             int AL_PayOff_As_Regressor)
{
  int i,j;
  long k;
  double tmp;
  PnlVect VStock;
  VStock.size=BS_Dimension;

  pnl_vect_set_double (AuxR, 0.);
  pnl_mat_set_double (M, 0.0);

  for(k=0;k<AL_MonteCarlo_Iterations;k++){
  /*value of the regressor basis on the kth path*/
  if (AL_PayOff_As_Regressor<=Time){
    /*here, the payoff function is introduced in the
       * regression basis*/
      VStock.array=&(Paths[k*BS_Dimension]);
      pnl_vect_set (VBase, 0, p->Compute(p->Par, &VStock));
      for (i=1;i<AL_Basis_Dimension;i++){
        pnl_vect_set (VBase, i, pnl_basis_i(Basis,Paths+k*
    BS_Dimension,i-1));
```

```
      }
    } else {
      for (i=0;i<AL_Basis_Dimension;i++){
        pnl_vect_set (VBase, i, pnl_basis_i(Basis,Paths+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, Vector_Price[k] * pnl_vect_get
     (VBase,i) + tmp);
    }
  }
  pnl_vect_clone (Res, AuxR);
  /* solve in the least square sense, using a QR decomposi
    tion */
  pnl_mat_ls (M, Res);
}


static void Close()
{
  /*memory liberation*/
  TsRoB_Liberation();
  End_BS();
  /*useful only when the payoff function is a formulae*/
}

/*see the documentation for the parameters meaning*/
int TsRoB(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_Exercise_Dates,
          int AL_PayOff_As_Regressor,
          double *AL_FPrice,
          double *AL_BPrice)
{
  double DiscountStep,Step,Aux,AuxOption;
  long i;
  int k, init_mc;
  int BS_Dimension =  BS_Spot->size;


  /* 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_Exercise_Dates-1);
  /*discounting factor for a time step*/
  DiscountStep=exp(-BS_Interest_Rate*Step);

  /*memory allocation of the BlackScholes variables*/
  Init_BS(BS_Dimension,BS_Volatility->array,
      BS_Correlation,BS_Interest_Rate,BS_Dividend_Rate->
    array);
  /*Initialization of the regression basis*/
  Basis = pnl_basis_create (name_basis, AL_Basis_Dimension,
     BS_Dimension);
  /*memory allocation of the algorithm's variables*/
  TsRoB_Allocation(AL_MonteCarlo_Iterations,AL_Basis_Dimens
    ion, BS_Dimension);
```

```
/*initialisation of the brownian bridge at the maturity*/
Init_Brownian_Bridge(Brownian_Bridge,AL_MonteCarlo_Itera
  tions,
           BS_Dimension,OP_Maturity, generator);
/*computation of the BlackScholes paths at the maturity
  related to Brownian_Bridge*/
Backward_Path(Paths,Brownian_Bridge,BS_Spot->array,OP_
  Maturity,
      AL_MonteCarlo_Iterations,BS_Dimension);
/*computation of the dynamical programming prices at
  time OP_Exercise_Dates-1*/
Compute_Vector_Price(AL_MonteCarlo_Iterations,p, OP_Exerc
  ise_Dates,AL_Basis_Dimension,BS_Dimension,
           OP_Exercise_Dates-1,AL_PayOff_As_Regres
  sor,DiscountStep);
for (k=OP_Exercise_Dates-2;k>=1;k--){
  /*computation of the brownian bridge at time k*/
  Compute_Brownian_Bridge(Brownian_Bridge,k*Step,Step,BS_
  Dimension,
                        AL_MonteCarlo_Iterations,     generator);
  /*computation of the BlackScholes paths at time k rel
  ated to Brownian_Bridge*/
  Backward_Path(Paths,Brownian_Bridge,BS_Spot->array,(
  double)k*Step,
              AL_MonteCarlo_Iterations,BS_Dimension);
  /*regression procedure*/
  Regression(AL_MonteCarlo_Iterations,OP_Exercise_Dates,
  p,
           AL_Basis_Dimension,BS_Dimension,k,AL_PayOff_
  As_Regressor);
  /*computation of the dynamical programming prices at
  time k*/
  Compute_Vector_Price(AL_MonteCarlo_Iterations,p, OP_Exe
  rcise_Dates,AL_Basis_Dimension,
                     BS_Dimension,k,AL_PayOff_As_Regres
  sor,DiscountStep);
}

Aux=0;
/*at time 0, the conditionnal expectation reduces to an
  expectation*/
```

```
  for (i=0;i<AL_MonteCarlo_Iterations;i++){
  Aux+=Vector_Price[i];
  }
  Aux/=(double)AL_MonteCarlo_Iterations;
  AuxOption=p->Compute(p->Par,BS_Spot);
  /*output backward price*/
  *AL_BPrice=MAX(AuxOption,Aux);

  Aux=0;
  for (i=0;i<AL_MonteCarlo_Iterations;i++){
  Aux+=FP[i];
  }
  Aux/=(double)AL_MonteCarlo_Iterations;

  /*output forward price*/
  *AL_FPrice=MAX(AuxOption,Aux);

  pnl_basis_free (&Basis);
  Close();
  return OK;
}

int CALC(MC_TsitsiklisVanRoyND)(void *Opt, void *Mod, Prici
    ngMethod *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_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;



  res=TsRoB(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_ENUM.value,
            Met->Par[3].Val.V_INT,
            Met->Par[4].Val.V_INT,
            Met->Par[5].Val.V_ENUM.value,
            &(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_TsitsiklisVanRoyND)(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_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.members=&PremiaEnumBool;
    }
  return OK;
}

PricingMethod MET(MC_TsitsiklisVanRoyND)=
{
  "MC_TsitsiklisVanRoy_ND",
  {{"N iterations",LONG,{100},ALLOW},
   {"RandomGenerator",ENUM,{100},ALLOW},
   {"Basis",ENUM,{1},ALLOW},
   {"Dimension Approximation",INT,{100},ALLOW},
   {"Number of Exercise Dates",INT,{100},ALLOW},
   {"Use Payoff as Regressor",ENUM,{1},ALLOW},
   {" ",PREMIA_NULLTYPE,{0},FORBID}},
  CALC(MC_TsitsiklisVanRoyND),
  {{"Forward Price",DOUBLE,{100},FORBID},
   {"Backward Price",DOUBLE,{100},FORBID},
   {" ",PREMIA_NULLTYPE,{0},FORBID}},
  CHK_OPT(MC_TsitsiklisVanRoyND),
  CHK_mc,
  MET(Init)
};
```

References