Help

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

#include "bsnd_stdnd.h"
#include "pnl/pnl_basis.h"
#include "optype.h"
#include "enums.h"
#include "pnl/pnl_random.h"
#include "pnl/pnl_matrix.h"

/* from #include "math/ImportanceSampling_jl/src/wrapper.hpp"
 * But as it is a C++ header, we cannot include it directly */
PnlVect* ComputeEuropeanBSDrift(PnlVect *spot, PnlVect *sig, PnlVect *divid, dou

/*
 * Parameters of the Black Scholes model
 */
typedef struct
{
  PnlVect *spot;
  PnlVect *sigma;
  PnlMat  *correl;
  double   r;
  PnlVect *divid;
  int      size;
  int      n_dates;
  double   dt;
  double   sqrt_dt;
  double   rho;
  PnlMat  *work;
} BSModel;

typedef struct _RegressorType RegressorType;
struct _RegressorType
{
  int size;
  int n_samples;
  int current_date;
```

```
  PnlMat **regressors;
  PnlBasis *basis;
  void (*computeCoefficients)(RegressorType *self, PnlVect *alpha, PnlVect *Y);
};

int onestepOMP(PnlVect *y_res, PnlVect *residu, PnlVect *coef, PnlVectInt *ind_c
{
  int k,Nb,L, gamma, test_inv;
  double a;
  int nb_coef=phi_t->m;
  Nb=phi->n; /* nb_func */
  L=phi->m; /* n_samples */

  PnlMat *phi_t_square = pnl_mat_new();
  PnlVect *phi_t_y = pnl_vect_new();
  PnlVect *phi_gamma = pnl_vect_new();
  PnlVect *phi_k=pnl_vect_create(L);
  PnlVect *y_phi=pnl_vect_create(Nb);

  pnl_vect_int_resize(ind_coeff,nb_coef+1);
  pnl_mat_resize(phi_t,nb_coef+1, L);
  for (k = 0 ; k < Nb ; k++)
    {
      pnl_mat_get_col(phi_k,phi,k);
      LET(y_phi,k)=fabs(pnl_vect_scalar_prod(residu,phi_k))/pnl_vect_norm_two(ph
    }

  pnl_vect_max_index(&a, &gamma, y_phi);
  LET_INT(ind_coeff, nb_coef) = gamma;
  pnl_mat_get_col(phi_gamma,phi,gamma);
  pnl_mat_set_row(phi_t,phi_gamma,nb_coef);

  pnl_mat_dgemm('N', 'T', 1., phi_t, phi_t, 0., phi_t_square);
  pnl_mat_mult_vect_inplace(phi_t_y, phi_t, y);
  test_inv = pnl_mat_syslin(coef, phi_t_square, phi_t_y);

  pnl_mat_mult_vect_transpose_inplace(y_res, phi_t, coef);
  pnl_vect_clone(residu,y_res);
  pnl_vect_minus_vect(residu,y);

  pnl_vect_free(&phi_k);
```

```
  pnl_vect_free(&y_phi);
  pnl_vect_free(&phi_gamma);
  pnl_mat_free(&phi_t_square);
  pnl_vect_free(&phi_t_y);

  return test_inv;
}

void computeCoefficientsOMP(RegressorType *self, PnlVect *coef, PnlVect *y)
{
  int i, j, k;
  PnlMat *phi, *phi_t;
  PnlVect *y_res, *coef_tmp, *residu, *coef_tmp_old;
  PnlVectInt *ind_coef;

  pnl_vect_resize(coef, self->basis->nb_func);
  pnl_vect_set_all(coef, 0.);
  phi = pnl_mat_create_from_scalar(self->n_samples,self->basis->nb_func, 0.);
  phi_t = pnl_mat_create_from_scalar(0, self->n_samples, 0.);
  ind_coef=pnl_vect_int_create_from_scalar(0,0);
  coef_tmp=pnl_vect_create_from_scalar(0,0);
  coef_tmp_old=pnl_vect_create_from_scalar(0,0);
  y_res=pnl_vect_create_from_scalar(self->n_samples,0);
  residu=pnl_vect_create_from_scalar(self->n_samples,0);
  pnl_vect_clone(residu,y);

  /* construct the matrix phi (of size n_samples x nb_func) */
  for (i = 0 ; i < self->n_samples ; i++)
    {
      PnlVect xi = pnl_vect_wrap_mat_row(self->regressors[i], self->current_date
      for (k = 0; k < self->basis->nb_func; k++)
        {
          MLET(phi,i,k) = pnl_basis_i_vect(self->basis, &xi, k);
        }
    }

  for (j = 0; j < self->basis->nb_func; j++)
    {
      pnl_vect_resize(coef_tmp_old, coef_tmp->size);
      if (onestepOMP(y_res, residu, coef_tmp, ind_coef, phi, phi_t, y) == OK)
        pnl_vect_clone(coef_tmp_old, coef_tmp);
```

```
      else
        break;
    }
  for (i = 0; i < coef_tmp_old->size; i++)
    {
      LET(coef, GET_INT(ind_coef, i)) = GET(coef_tmp_old, i);
    }

  pnl_vect_free(&y_res);
  pnl_vect_int_free(&ind_coef);
  pnl_mat_free(&phi);
  pnl_vect_free(&coef_tmp);
  pnl_vect_free(&coef_tmp_old);
  pnl_vect_free(&residu);
  pnl_mat_free(&phi_t);
}

void onestepMPP(PnlVect *y_res, PnlVect *residu, PnlVect *coef, PnlMat *phi)
{
  int k, Nb, L, gamma;
  double a;
  Nb = phi->n;
  L = phi->m;

  PnlVect *phi_gamma = pnl_vect_create_from_scalar(L, 0);
  PnlVect *phi_k = pnl_vect_create_from_scalar(L, 0);
  PnlVect *y_phi = pnl_vect_create_from_scalar(Nb, 0);

  for (k = 0 ; k < Nb ; k++)
    {
      pnl_mat_get_col(phi_k, phi, k);
      LET(y_phi, k) = fabs(pnl_vect_scalar_prod(residu, phi_k)) / pnl_vect_norm_
    }
  pnl_vect_max_index(&a, &gamma, y_phi);
  pnl_mat_get_col(phi_gamma, phi, gamma);
  LET(coef, gamma) = pnl_vect_scalar_prod(residu, phi_gamma) / pow(pnl_vect_norm
  pnl_vect_axpby(GET(coef, gamma), phi_gamma, 1., y_res);
  pnl_vect_minus_vect(residu, phi_gamma);

  pnl_vect_free(&phi_k);
  pnl_vect_free(&y_phi);
```

```
  pnl_vect_free(&phi_gamma);
}

void computeCoefficientsMPP(RegressorType *self, PnlVect *coef, PnlVect *y)
{
  int i, k;
  double n0, n1;
  PnlMat *phi;
  PnlVect *y_res, *residu;

  double reduction;
  int j = 0;
  pnl_vect_resize(coef, self->basis->nb_func);
  pnl_vect_set_all(coef, 0.);
  phi = pnl_mat_create_from_scalar(self->n_samples, self->basis->nb_func, 0.);
  y_res = pnl_vect_create_from_scalar(self->n_samples, 0);
  residu = pnl_vect_create_from_scalar(self->n_samples, 0);
  pnl_vect_clone(residu, y);
  reduction = pnl_vect_norm_two(residu);

  /* construct the matrix phi (of size self->n_samples x self->basis->nb_func) *
  for (i = 0 ; i < self->n_samples ; i++)
    {
      PnlVect xi = pnl_vect_wrap_mat_row(self->regressors[i], self->current_date
      for (k = 0 ; k < self->basis->nb_func ; k++)
        {
          MLET(phi, i, k) = pnl_basis_i_vect(self->basis, &xi, k);
        }
    }

  for (j = 0; j < self->basis->nb_func; j++)
    {
      if (reduction < 0.01) break;
      n0 = pnl_vect_norm_two(residu);
      onestepMPP(y_res, residu, coef, phi);
      n1 = pnl_vect_norm_two(residu);
      reduction = (n0 - n1);
    }

  pnl_vect_free(&y_res);
  pnl_vect_free(&residu);
```

```
  pnl_mat_free(&phi);
}


/**
 * Sample one path of the BS model
 *
 * @param[out] path holds a sample path of the model on output, n_dates x size
 * @param mod a BSModel
 * @param G the standard Brownian increments with size n_dates x size
 */
static void BSpath(PnlMat *path, BSModel *mod, const PnlMat *G)
{
  pnl_mat_resize(path, mod->n_dates + 1, mod->size);
  pnl_mat_set_row(path, mod->spot, 0);
  pnl_mat_resize(mod->work, mod->n_dates, mod->size);
  pnl_mat_dgemm('N', 'T', mod->sqrt_dt, G, mod->correl, 0., mod->work);
  for (int j = 0; j < mod->n_dates; j++)
    {
      for (int i = 0; i < mod->size; i++)
        {
          MLET(path, j + 1, i) = MGET(path, j, i)
           * exp((mod->r - GET(mod->divid, i) - GET(mod->sigma, i) * GET(mod->si
                 + GET(mod->sigma, i) * MGET(mod->work, j, i));
        }
    }
}


/**
 * Wrapper around the NumFunc_nd payoff function for use into PnlBasis
 *
 * @param x the spot as a PnlVect
 * @param p a NumFunc_nd pointer seen as a void *
 */
static double payoff(const PnlVect *x, void *p)
{
  NumFunc_nd *F = (NumFunc_nd *) p;
  return F->Compute(F->Par, x);
}
```

```
/**
 * Compute the Girsanov martingale at time t = n dt
 *
 * @param n current date as an integer
 * @param DeltaB the matrix of standard increments (n_dates x size) for the new
 * Brownian.
 * @param drift the drift vector, with size size. The reduced approach is used.
 */
double gaussian_weight(int n, PnlMat *DeltaB, PnlVect *drift)
{
  int i;
  double w = 0.;
  double norm = pnl_vect_scalar_prod(drift, drift) * n;
  for (i = 0; i < n; ++i)
    {
      PnlVect DeltaB_i = pnl_vect_wrap_mat_row(DeltaB, i);
      w += pnl_vect_scalar_prod(&DeltaB_i, drift);
    }
  /* Remember that DeltaB already contains the shift --- it is the new Brownian,
  return  exp(-w + norm / 2);
}

/**
 * Compute the Colecky factorization of the correlation matrix
 *
 * @param rho common correlation between two assets
 * @param size number of assets
 */
static PnlMat* compute_sqrt_correlation(double rho, int size)
{
  PnlMat *cov = pnl_mat_create(size, size);
  pnl_mat_set_all(cov, rho);
  pnl_mat_set_diag(cov, 1., 0);
  pnl_mat_chol(cov);
  return cov;
}


/**
 * Map an upperbound of the domanin to [-1,1]^size
```

```
 *
 * @param B
 * @param mod
 */
void set_domain_bounds (PnlBasis *B, const BSModel *mod)
{
  int i;
  PnlVect *min, *max;
  double T = mod->n_dates * mod->dt;
  min = pnl_vect_create (mod->size);
  max = pnl_vect_create (mod->size);
  for ( i=0 ; i<mod->size ;i++ )
    {
      double sig = GET(mod->sigma, i);
      LET(min, i) = GET(mod->spot,i) * exp ((mod->r - sig*sig/2) * T - sig * 4.0
      LET(max, i) = GET(mod->spot,i) * exp ((mod->r - sig*sig/2) * T + sig * 4.0
    }
  pnl_basis_set_domain(B, min, max);
  pnl_vect_free (&min);
  pnl_vect_free (&max);
}


/**
 * Solve the regression problem.
 *
 * @param self an instance of RegressorType.
 * @param[out] alpha the coefficients of the regression on output.
 * @param Y the values to be explained.
 */
void computeCoefficientsLeastQuare(RegressorType *self, PnlVect *alpha, PnlVect
{
  int i, k;
  pnl_vect_resize(alpha, self->basis->nb_func);
  pnl_vect_set_zero(alpha);
  PnlVect *phi_k = pnl_vect_create_from_scalar(self->basis->nb_func, 0.);
  PnlMat *A = pnl_mat_create_from_scalar(self->basis->nb_func, self->basis->nb_f

  /* Construct A and b, to solve A * alpha = b*/
  for (i = 0 ; i < self->n_samples ; i++)
    {
```

```
      PnlVect xi = pnl_vect_wrap_mat_row(self->regressors[i], self->current_date
      for (k = 0 ; k < self->basis->nb_func ; k++)
        {
          const double tmp = pnl_basis_i_vect(self->basis, &xi, k);
          double b_k =  PNL_GET(alpha, k);
          b_k += tmp * PNL_GET(Y, i);
          PNL_LET(alpha, k) = b_k;
          PNL_LET(phi_k, k) = tmp;
        }
      /* A += phi_k' * phi_k */
      pnl_mat_dger(1., phi_k, phi_k, A);
    }

  pnl_mat_ls(A, alpha);
  pnl_vect_free(&phi_k);
  pnl_mat_free(&A);
}


/**
 * Solve the dynamic programming principle
 *
 * @param mod the BS model
 * @param n_samples number of MC samples
 * @param p_numfunc the instance of the NumFunc_nd pointer for the current optio
 * @param basis the family of functions to regress upon
 * @param p_S an array of sample path from the spot, n_samples x ((n_dates +1) x
 * @param DeltaB an array of Brownian increments, n_samples x (n_dates x size)
 * @param p_Ztau the vector of discounted payoffs at the optimal stopping time c
 * @param reg the regressor
 * @param use_is Use Importance Sampling. The optimal drift comes from the corre
 */
void solve_backward_induction(BSModel *mod, int n_samples, NumFunc_nd *p_numfunc
{
  int l;
  int T = mod->n_dates;
  PnlVect *beta = pnl_vect_new();

  for (int t = T - 1; t > 0; --t)
    {
      /* Regression step */
```

```
      reg->current_date = t;
      reg->computeCoefficients(reg, beta, p_Ztau);

      /* Update each path */
      for (l = 0; l < n_samples; l++)
        {
          PnlVect St = pnl_vect_wrap_mat_row(p_S[l], t);
          double value = payoff(&St, p_numfunc) * exp(-mod->r * t * mod->dt);
          if (use_is) value *= gaussian_weight(t, DeltaB[l], drift);

          if (value == 0.) continue;
          double Et = pnl_basis_eval_vect(basis, beta, &St);
          if (value > 0. && value > Et)
            {
              LET(p_Ztau, l) = value;
            }
        }
    }
  pnl_vect_free(&beta);
}



/**
 * Run the policy iteration algorithm for American option (Longstaff Schwartz ap
 *
 * @param mod the BS model
 * @param[out] ptprix holds the price on output
 * @param n_samples number of MC samples
 * @param rng_index the name of the PnlRng to be used
 * @param use_anti Use antithetic sample paths. Can be 0 or 1
 * @param use_is Use Importance Sampling. The optimal drift comes from the corre
 * @param p_numfunc the instance of the NumFunc_nd pointer for the current optio
 * @param basis the family of functions to regress upon
 */
static void longstaff_schwartz(BSModel *mod, double *ptprix, int n_samples, int
{
  int l;
  RegressorType Regressor;
  PnlMat **DeltaB = malloc(n_samples * sizeof(PnlMat*));
  PnlMat **S = malloc(n_samples * sizeof(PnlMat*));
```

```
PnlVect *Ztau = pnl_vect_create(n_samples);
PnlRng *rng = pnl_rng_create(rng_index);
PnlVect *drift = NULL;
int T = mod->n_dates;
pnl_rng_sseed(rng, 0);

if (use_is)
  {
    drift =  ComputeEuropeanBSDrift(mod->spot, mod->sigma, mod->divid, mod->r,
    pnl_vect_mult_scalar(drift, mod->sqrt_dt);
  }

if (use_anti)
  {
    n_samples = (n_samples / 2) * 2;
    int n_samples_2 = n_samples / 2;
    for (l = 0; l < n_samples_2; l++)
      {
        DeltaB[l] = pnl_mat_create(T, mod->size);
        pnl_mat_rng_normal(DeltaB[l], T, mod->size, rng);
        DeltaB[l + n_samples_2] = pnl_mat_copy(DeltaB[l]);
        pnl_mat_mult_scalar(DeltaB[l + n_samples_2], -1.);
      }
  }
else
  {
    for (l = 0; l < n_samples; l++)
      {
        DeltaB[l] = pnl_mat_create(T, mod->size);
        pnl_mat_rng_normal(DeltaB[l], T, mod->size, rng);
      }
  }

/*
 * Initialization at maturity time
 */
for (l = 0; l < n_samples; l++)
  {
    S[l] = pnl_mat_create(T + 1, mod->size);

    /* Add the drift to all Brownian increments */
```

```
    if (use_is)
      {
        int j;
        for (j = 0; j < mod->n_dates; ++j)
          {
            PnlVect dB = pnl_vect_wrap_mat_row(DeltaB[l], j);
            pnl_vect_plus_vect(&dB, drift);
          }
      }
    BSpath(S[l], mod, DeltaB[l]);
    PnlVect St = pnl_vect_wrap_mat_row(S[l], T);
    LET(Ztau, l) = payoff(&St, p_numfunc) * exp(-mod->r * T * mod->dt);
    if (use_is) LET(Ztau, l) *= gaussian_weight(T, DeltaB[l], drift);
  }

Regressor.n_samples = n_samples;
switch (regression_method)
  {
  case 1: /* Original LS method */
    Regressor.regressors = S;
    Regressor.basis = basis;
    Regressor.computeCoefficients = computeCoefficientsLeastQuare;
    break;
  case 2: /* Matching Pursuit */
    Regressor.regressors = S;
    Regressor.basis = basis;
    Regressor.computeCoefficients = computeCoefficientsMPP;
    break;
  case 3: /* Orthogonal Matching Pursuit */
    Regressor.regressors = S;
    Regressor.basis = basis;
    Regressor.computeCoefficients = computeCoefficientsOMP;
    break;
  default:
    printf("Unknown regression method: %d\ n", regression_method);
    abort();
  }

solve_backward_induction(mod, n_samples, p_numfunc, basis, S, DeltaB, Ztau, &R

*ptprix = pnl_vect_sum(Ztau) / n_samples;
```

```
  // Exercise at time 0?
  double prix_0 = payoff(mod->spot, p_numfunc);
  if (prix_0 > *ptprix) { *ptprix = prix_0; }

  pnl_vect_free(&Ztau);
  pnl_rng_free(&rng);
  pnl_vect_free(&drift);
  for (l = 0; l < n_samples; l++)
    {
      pnl_mat_free(&DeltaB[l]);
      pnl_mat_free(&S[l]);
    }
  free(DeltaB);
  free(S);
}


int CALC(MC_LongstaffSchwartzND)(void *Opt, void *Mod, PricingMethod *Met)
{
  TYPEOPT *ptOpt = (TYPEOPT *)Opt;
  TYPEMOD *ptMod = (TYPEMOD *)Mod;
  double rho;
  int size, i;
  PnlVect *divid, *spot, *sig;
  PnlMat *correl;
  PnlBasis *basis;
  PnlRnFuncR F;
  BSModel bsmod;

  spot = pnl_vect_compact_to_pnl_vect(ptMod->S0.Val.V_PNLVECTCOMPACT);
  sig = pnl_vect_compact_to_pnl_vect(ptMod->Sigma.Val.V_PNLVECTCOMPACT);
  divid = pnl_vect_compact_to_pnl_vect(ptMod->Divid.Val.V_PNLVECTCOMPACT);
  size = ptMod->Size.Val.V_PINT;
  rho = ptMod->Rho.Val.V_DOUBLE;

  for (i = 0; i < size; i++)
    LET(divid, i) = log(1. + GET(divid, i) / 100);

  if (rho > 1 || rho < -1. / (size - 1))
    {
      perror("correlation out of range");
```

```
    return FAIL;
  }
correl = compute_sqrt_correlation(rho, ptMod->Size.Val.V_PINT);

bsmod.spot = spot;
bsmod.sigma = sig;
bsmod.rho = rho;
bsmod.correl = correl;
bsmod.r = log(1. + ptMod->R.Val.V_DOUBLE / 100.);
bsmod.divid = divid;
bsmod.size = size;
bsmod.n_dates = Met->Par[4].Val.V_INT;
bsmod.dt = (ptOpt->Maturity.Val.V_DATE - ptMod->T.Val.V_DATE) / bsmod.n_dates;
bsmod.sqrt_dt = sqrt(bsmod.dt);
bsmod.work = pnl_mat_new();

basis = pnl_basis_create_from_degree(Met->Par[2].Val.V_ENUM.value, Met->Par[3]
set_domain_bounds(basis, &bsmod);

if (Met->Par[6].Val.V_ENUM.value == 1)
  {
    F.F = payoff;
    F.params = ptOpt->PayOff.Val.V_NUMFUNC_ND;
    pnl_basis_add_function(basis, &F);
  }
longstaff_schwartz(&bsmod,
                   &(Met->Res[0].Val.V_DOUBLE),
                   Met->Par[0].Val.V_LONG,
                   Met->Par[1].Val.V_ENUM.value,
                   Met->Par[7].Val.V_ENUM.value,
                   Met->Par[8].Val.V_ENUM.value,
                   Met->Par[5].Val.V_ENUM.value,
                   ptOpt->PayOff.Val.V_NUMFUNC_ND, basis);

pnl_vect_free(&divid);
pnl_vect_free(&spot);
pnl_vect_free(&sig);
pnl_mat_free(&correl);
pnl_basis_free(&basis);
pnl_mat_free(&(bsmod.work));
return OK;
```

```
}

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;
  else
    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 = PNL_BASIS_HERMITE;
      Met->Par[2].Val.V_ENUM.members = &PremiaEnumBasis;
      Met->Par[3].Val.V_INT = 3;
      Met->Par[4].Val.V_INT = 10;
      Met->Par[5].Val.V_ENUM.value = 1;
      Met->Par[5].Val.V_ENUM.members = &PremiaEnumRegressionType;
      Met->Par[6].Val.V_ENUM.value = 1;
      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, SETABLE, NULL},
```

```
    {"RandomGenerator", ENUM, {0}, ALLOW, SETABLE, NULL},
    {"Basis", ENUM, {1}, ALLOW, SETABLE, NULL},
    {"Degree of Approximation", INT, {100}, ALLOW, SETABLE, NULL},
    {"Number of Exercise Dates", INT, {100}, ALLOW, SETABLE, NULL},
    {"Regression type", ENUM, {1}, ALLOW, SETABLE, NULL},
    {"Use Payoff as Regressor", ENUM, {1}, ALLOW, SETABLE, NULL},
    {"Use Antithetic Variables", ENUM, {1}, ALLOW, SETABLE, NULL},
    {"Use Importance Sampling", ENUM, {0}, ALLOW, SETABLE, NULL},
    {" ", PREMIA_NULLTYPE, {0}, FORBID, UNSETABLE, NULL}
  },
  CALC(MC_LongstaffSchwartzND),
  { {"Price", DOUBLE, {100}, FORBID, SETABLE, NULL},
    {" ", PREMIA_NULLTYPE, {0}, FORBID, SETABLE, NULL}
  },
  CHK_OPT(MC_LongstaffSchwartzND),
  CHK_mc,
  MET(Init),
  0,
  "mc_longstaffschwatrz_nd"
};
```