Help

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "pnl/pnl_vector.h"
#include "pde_tools.h"


/**
 * creates a PremiaPDEBoundary
 * @param X0 down_left point of domain
 * @param X1 Up_right point of domain
 * @return a PremiaPDEBoundary pointer
 */
PremiaPDEBoundary premia_pde_boundary_create(double X0, double X1)
{
  PremiaPDEBoundary  BP;
  BP.X0 = X0;
  BP.H = 1 / (X1 - X0);
  return BP;
}

double premia_pde_boundary_real_variable(const PremiaPDEBoundary BP , double X)
{
  return  BP.X0 + X / BP.H;
}

double premia_pde_boundary_unit_interval(const PremiaPDEBoundary BP , double X)
{
  double res = (X - BP.X0) * BP.H;
  if (fabs(2 * res - 1) <= 1.0)
    {
      printf("error on boundary Unit_Interval");
      abort();
    }
  return res;
}

/**
 * creates a PremiaPDEBoundary with
```

```
 * Left down corner is \ f$ (0,\ dots,0)\ f$
 * and right up corner is \ f$ (1,\ dots,1)\ f$
 * @return a PremiaPDEDimBoundary pointer
 */
PremiaPDEDimBoundary *premia_pde_dim_boundary_create_from_int(int dim)
{
  PremiaPDEDimBoundary *TBP;
  PremiaPDEBoundary *Tmp;

  int i;
  if ((TBP = malloc(sizeof(PremiaPDEDimBoundary))) == NULL)
    return NULL;
  if (dim > 0)
    {
      if ((TBP->array = malloc(dim * sizeof(PremiaPDEBoundary))) == NULL)
        return NULL;
    }
  else
    TBP->array = (PremiaPDEBoundary *)NULL;
  Tmp = TBP->array;
  i = 0;
  while (i < dim)
    {
      (*Tmp) = premia_pde_boundary_create(0.0, 1.0);
      Tmp++;
      i++;
    }
  return TBP;
}

/**
 * creates a PnlMat
 * @param X0 left down corner
 * @param X1 right up corner
 * @return a PremiaPDEDimBoundary pointer
 */
PremiaPDEDimBoundary *premia_pde_dim_boundary_create(const PnlVect *X0,
    const PnlVect *X1)
{
  PremiaPDEDimBoundary *TBP;
  PremiaPDEBoundary *Tmp;
```

```
  int i;
  if ((TBP = malloc(sizeof(PremiaPDEDimBoundary))) == NULL)
    return NULL;
  if (X0->size > 0)
    {
      if ((TBP->array = malloc(X0->size * sizeof(PremiaPDEBoundary))) == NULL)
        return NULL;
    }
  else
    TBP->array = (PremiaPDEBoundary *)NULL;
  Tmp = TBP->array;
  i = 0;
  while (i < X0->size)
    {
      (*Tmp) = premia_pde_boundary_create(GET(X0, i), GET(X1, i));
      Tmp++;
      i++;
    }
  return TBP;
}

/**
 * frees a PremiaPDEDimBoundary
 *
 * @param v adress of a PremiaPDEDimBoundary*. v is set to NULL at exit.
 */
void premia_pde_dim_boundary_free(PremiaPDEDimBoundary **v)
{

  if (*v != NULL)
    {
      if ((*v)->array != NULL)
        {
          free((*v)->array);
          free(*v);
          *v = NULL;
        }
    }
}

double premia_pde_dim_boundary_eval_from_unit(double(*f)(const PnlVect *),
```

```
    const PremiaPDEDimBoundary *BP,
    const PnlVect *X)
{

  PnlVect *Res;
  double sol;
  int i;
  Res = pnl_vect_create(X->size);
  for (i = 0; i < X->size; i++)
    LET(Res, i) = premia_pde_boundary_real_variable(BP->array[i], GET(X, i));
  sol = f(Res);
  pnl_vect_free(&Res);
  return  sol;
}

void premia_pde_dim_boundary_from_unit_to_real_variable(const PremiaPDEDimBounda
    PnlVect *X)
{
  int i;
  for (i = 0; i < X->size; i++)
    LET(X, i) = premia_pde_boundary_real_variable(BP->array[i], GET(X, i));
}

double premia_pde_dim_boundary_get_step(const PremiaPDEDimBoundary *BP,
                                        int i)
{
  return BP->array[i].H;
}




double standard_time_repartition(int i, int N_T)
{
  return (double)(i) / (double) N_T;
}
/**
 * creates a PremiaPDETimeGrid
 * @param T Terminal time
 * @param N_T number of grids points
```

```
 * @param repartition function for repartitions of grids points
 * @return a PremiaPDETimeGrid pointer
 */
PremiaPDETimeGrid *premia_pde_time_grid(const double T,
                                        const int N_T,
                                        double (*repartition)(int i, int NN)
                                       )
{
  PremiaPDETimeGrid  *TG;
  int i;
  if ((TG = malloc(sizeof(PremiaPDETimeGrid))) == NULL)
    return NULL;
  if (N_T > 0)
    {
      TG->time = pnl_vect_create(N_T + 1);
      i = 0;
      do
        {
          LET(TG->time, i) = T * repartition(i, N_T);
          i++;
        }
      while (i <= N_T);
      TG->is_tuned = 1;
      premia_pde_time_start(TG);
    }
  else
    TG->time = (PnlVect *)NULL;
  return TG;
}
/**
 * creates a PremiaPDETimeGrid
 * @param T Terminal time
 * @param N_T number of grids points
 * @return a PremiaPDETimeGrid pointer
 */
PremiaPDETimeGrid *premia_pde_time_homogen_grid(const double T,
    const int N_T)
{
  PremiaPDETimeGrid  *TG = premia_pde_time_grid(T, N_T, standard_time_repartitio
  TG->is_tuned = 0;
  return TG;
```

```
}



/**
 * frees a PremiaPDETimeGrid
 *
 * @param TG adress of a PremiaPDETimeGrid*. TG is set to NULL at exit.
 */
void premia_pde_time_grid_free(PremiaPDETimeGrid **TG)
{

  if (*TG != NULL)
    {
      pnl_vect_free(&((*TG)->time));
      free(*TG);
      *TG = NULL;
    }
}

/**
 * initialise PremiaPDETimeGrid to the first step.
 *
 * @param TG a PremiaPDETimeGrid pointer
 * initialise in first step
 */
void premia_pde_time_start(PremiaPDETimeGrid *TG)
{
  TG->current_index = 0;
  TG->current_step = GET(TG->time, 1) - GET(TG->time, 0);
  /*-GET(TG->time,TG->current_index)+GET(TG->time,++(TG->current_index)); */
}

/**
 * go to the next time step
 *
 * @param TG a PremiaPDETimeGrid pointer
 * increase the current time and compute current step
 * @return 0 if last time step
 */
int premia_pde_time_grid_increase(PremiaPDETimeGrid *TG)
```

```
{
  if (TG->is_tuned == 1)
    TG->current_step = -GET(TG->time, TG->current_index) + GET(TG->time, TG->cur
  (TG->current_index) += 1;
  return (TG->current_index < TG->time->size);
}


/**
 * GET function on current step
 *
 * @param TG a PremiaPDETimeGrid pointer
 * @return the current step
 */
double premia_pde_time_grid_step(const PremiaPDETimeGrid *TG)
{
  return TG->current_step;
}

/**
 * GET function on current time
 *
 * @param TG a PremiaPDETimeGrid pointer
 * @return the current time
 */
double premia_pde_time_grid_time(const PremiaPDETimeGrid *TG)
{
  return GET(TG->time, TG->current_index);
}
```