Help

```
/*
 * Writen by David Pommier <david.pommier@gmail.com>
 * INRIA 2009
 */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "pnl/pnl_random.h"
#include "pnl/pnl_vector_int.h"
#include "pnl/pnl_linalgsolver.h"
#include "pnl/pnl_finance.h"
#include "pde_tools.h"
#include "gridsparse_functions.h"
#include "gridsparse_constructor.h"


#define SPARSE_N2H(Vout,Vin,index,father)  LET(Vout,index)=
    GET(Vin,index)-0.5*( GET(Vin,(*father))+GET(Vin,*(father+1))
    )
#define SPARSE_H2N(Vout,Vin,index,father)  LET(Vout,index)=
    GET(Vin,index)+0.5*( GET(Vout,(*father))+GET(Vout,*(father+1
    )))
#define SPARSE_N2H_FUNC(Vin,index,father)  GET(Vin,index)-0
    .5*( GET(Vin,(*father))+GET(Vin,*(father+1)))


int log2int(int x)
{
  int level, y;
  if(x==0){printf("error in log2int ");abort();}
  level=0;
  y=x;
  while(y>1)
    {
      level++;
      y>>=1;
    }
  return level;
}
```

```
static double dyadic_cast(int i)
{
  /* {frac{2 (i-2^(lev))+1 }{2^{lev+1}} */
  int l=log2int(i);
  return (double)(((i-(1<<l))<<1)+1)/
    (double)(2<<l);}

static void vect_int_dyadic_cast(const PnlVectInt * v_in,Pn
    lVect * v_out)
{
  int i;
  for(i=0;i<v_in->size;i++)
    LET(v_out,i)=dyadic_cast(v_in->array[i]);
}




double GridSparse_real_value_at_points(GridSparse *G,int d,
    int i)
{
  return premia_pde_boundary_real_variable(G->Bnd->array[d]
    ,
                                    dyadic_cast(pnl_
    mat_int_get(G->Points,i,d)));
}

/*static double Nodal_to_Hier_in_point(const int  i,const
    int Dir,
  const PnlVect * V,
  const GridSparse * G)
  {
  int PP[3]={Dir,i,0};
  int * father = pnl_hmat_int_lget(G->Ind_Father,PP);
  return GET(V,i)-0.5*( GET(V,(*father))+GET(V,*(++father))
    );
  };*/

/**
 * Compute hierarchical coefficients from nodal coefficient
```

```
    s
 * in one direction
 *
 * @param  Dir a int, the direction,
 * @param V a PnlVect pointer on nodal coefficients,
 * @param Vout a PnlVect pointer on hierarchic coefficient
    s,
 * @param G a GridSparse pointer
 */
void Nodal_to_Hier_in_dir(const int Dir, const PnlVect  *V,
                          PnlVect *Vout,  const GridSparse
    * G)
{

  if (Dir>G->dim) {printf("error in dimension");abort();}
  else
    {
      int i=1;
      int PP[3]={Dir,1,0};
      int * father = pnl_hmat_int_lget(G->Ind_Father,PP);
      do
  {
    LET(Vout,i)=GET(V,i)-0.5*( GET(V,(*father))+GET(V,*(
    father+1)));
    i++;father+=2;
  }while(i<G->size);
    }
};


/**
 * Compute nodal coefficients from hierarchical coefficient
    s
 * in one direction
 *
 * @param  Dir a int, the direction,
 * @param V a PnlVect pointer on  hierarchic coefficients,
 * @param Vout a PnlVect pointer on nodal coefficients,
 * @param G a GridSparse pointer
 */
void Hier_to_Nodal_in_dir(int Dir, const PnlVect  * V, PnlV
```

```
    ect * Vout,  const GridSparse * G)
{
  if (Dir>G->dim) {printf("error in dimension");abort();}
  else
    {
      int i=1;
      int PP[3]={Dir,i,0};
      int * father = pnl_hmat_int_lget(G->Ind_Father,PP);
      do
  {
    LET(Vout,i)=GET(V,i)+0.5*( GET(Vout,(*father))+GET(Vou
    t,*(father+1)));
    i++;father+=2;
  }while(i<G->size);
    }
}


/**
 * Compute hierarchical coefficients from nodal coefficient
    s
 *
 * @param V a PnlVect pointer of nodalcoefficients
 * which becomes hierarchic coefficients,
 * @param G a GridSparse pointer
 */
void Nodal_to_Hier(PnlVect * V,const GridSparse * G)
{
  int i;
  PnlVect * V_Tmp=pnl_vect_copy(V);
  if(G->dim % 2 == 0)
    for(i=0;i<G->dim;i++)
      {
        Nodal_to_Hier_in_dir(i,V,V_Tmp,G);
        i++;
        Nodal_to_Hier_in_dir(i,V_Tmp,V,G);
      }
  else
    {
      for(i=0;i<G->dim-1;i++)
        {
```

```
        Nodal_to_Hier_in_dir(i,V_Tmp,V,G);
        i++;
        Nodal_to_Hier_in_dir(i,V,V_Tmp,G);
      }
    Nodal_to_Hier_in_dir(G->dim-1,V_Tmp,V,G);
  }
  pnl_vect_free(&V_Tmp);
}

/**
 * Compute nodal coefficients from hierarchical coefficient
    s
 *
 * @param V a PnlVect pointer of hierarchic coefficients
 * which becomes nodal coefficients,
 * @param G a GridSparse pointer
 */
/*
  void Hier_to_Nodal(PnlVect* V,const GridSparse *G)
  {
  PnlVect * V_Tmp= pnl_vect_copy(V);
  int i;
  if(G->dim % 2 == 0)
  for(i=G->dim-1;i>0;i--)
  {
  Hier_to_Nodal_in_dir(i,V,V_Tmp,G);
  i--;
  Hier_to_Nodal_in_dir(i,V_Tmp,V,G);
  }
  else
  {
  for(i=G->dim-1;i>1;i--)
  {
  Hier_to_Nodal_in_dir(i,V_Tmp,V,G);
  i--;
  Hier_to_Nodal_in_dir(i,V,V_Tmp,G);
  }
  Hier_to_Nodal_in_dir(0,V_Tmp,V,G);
  }
  pnl_vect_free(&V_Tmp);
  };
```

```
*/
void Hier_to_Nodal(PnlVect* V,const GridSparse *G)
{
  int i;
  PnlVect * V_Tmp= pnl_vect_copy(V);
  if(G->dim % 2 == 0)
    for(i=0;i<G->dim-1;i++)
      {
        Hier_to_Nodal_in_dir(i,V,V_Tmp,G);
        i++;
        Hier_to_Nodal_in_dir(i,V_Tmp,V,G);
      }
  else
    {
      for(i=0;i<G->dim-1;i++)
        {
          Hier_to_Nodal_in_dir(i,V_Tmp,V,G);
          i++;
          Hier_to_Nodal_in_dir(i,V,V_Tmp,G);
        }
      Hier_to_Nodal_in_dir(G->dim-1,V_Tmp,V,G);
    }
  pnl_vect_free(&V_Tmp);
}

/**
 * Compute nodal coefficients from hierarchical coefficient
    s
 *
 * @param Vin a PnlVect pointer of hierarchic coefficients
 * @param G a GridSparse pointer
 * @return a PnlVect pointer with nodal coefficients,
 */
PnlVect * V_Hier_to_Nodal(const PnlVect * Vin,
                          const GridSparse * G)
{
  PnlVect * VTmp = pnl_vect_copy(Vin);
  Hier_to_Nodal(VTmp,G);
  return VTmp;
}
```

```
/**
 * Compute distance between two points in dyadic represent
   ation
 *
 * @param a a Int
 * @param b a Int
 * @return abs(abcisse(a)-abcisse(b))
 */
static int Step(int a,int b)
{ return (a<b)?log2int(b)+1:log2int(a)+1;}

/**
 * A Get function,
 *
 * @param G a GridSparse pointer
 * @param dir a int, the direction,
 * @return size of domaine in direction dir.
 */
static double Domain_Size(const GridSparse * G,int dir)
{return G->Bnd->array[dir].H;}

/**
 * WARNING:Assumption homogen Grid,
 * Compute the second finite difference operator in direc
   tion dir, on point i
 *
 * @param i a int, index of point on which we compute fini
   te differencial operator
 * @param dir a int, direction in which we compute finite
   differencial operator
 * @param v a PnlVect pointer of nodal coefficients in dir
   ection dir
 * and hierarchic coefficients in others directions
 * @param G a GridSparse pointer
 * @param l_Ind_neig an int, index of left neighbour
 * @param r_Ind_neig an int, index of right neighbour
 * @return a double equal to the discret finite difference operator
 */
double FD_Lap_Stencil_Center(const int i,const int dir,
                             const PnlVect * v,
                             GridSparse * G,
```

```
                             int l_Ind_neig,
                             int r_Ind_neig)
{
  double Res =GET(v,(l_Ind_neig))-2.*GET(v,i)+GET(v,r_Ind_
    neig);
  Res*=pow(Domain_Size(G,dir),2)
    *(double)(1<<(2*(Step(pnl_mat_int_get(G->Points,i,dir),
                          pnl_mat_int_get(G->Points,l_Ind_
    neig,dir)))));
  /*  if ((l_Ind_neig==0)||(r_Ind_neig==0))
      {
      Res =GET(v,(l_Ind_neig))-2*GET(v,i)+GET(v,r_Ind_neig)
    ;
      Res*=pow(Domain_Size(G,dir),2)
      *(double)(1<<(2*(Step(pnl_mat_int_get(G->Points,i,dir
    ),
      pnl_mat_int_get(G->Points,l_Ind_neig,dir)))));
      printf(" point %d, index %d steps %d val %7.4f, val %
    7.4f --> Value %7.4f{n ",i,pnl_mat_int_get(G->Points,i,dir
    )
      ,Step(pnl_mat_int_get(G->Points,i,dir),
      pnl_mat_int_get(G->Points,l_Ind_neig,dir)),
      GET(v,l_Ind_neig),+GET(v,r_Ind_neig),Res);

      }*/
  return Res;
}

/**
 * WARNING:Assumption homogen Grid,
 * Compute the first order finite difference operator with
    centered scheme
 * in direction dir, on point i
 *
 * @param i a int, index of point on which we compute fini
    te differencial operator
 * @param dir a int, direction in which we compute finite
    differencial operator
 * @param v a PnlVect pointer of nodal coefficients in dir
    ection dir
 * and hierarchic coefficients in others directions
```

```
 * @param G a GridSparse pointer
 * @param l_Ind_neig an int, index of left neighbour
 * @param r_Ind_neig an int, index of right neighbour
 * @return a double equal to the discret finite difference operator
 */
double FD_Conv_Stencil_Center(const int i,const int dir,
                              const PnlVect * v,
                              const GridSparse * G,
                              int l_Ind_neig,
                              int r_Ind_neig)
{
  double res =GET(v,r_Ind_neig)-GET(v,l_Ind_neig);
  res*=Domain_Size(G,dir)*
    (double)(1<<(Step(pnl_mat_int_get(G->Points,i,dir),
                      pnl_mat_int_get(G->Points,l_Ind_neig,
    dir))-1));
  return res;
}

/**
 * WARNING:Assumption homogen Grid,
 * Compute the first order finite difference operator with
    centered scheme in direction dir, on point i
 *
 * @param i a int, index of point on which we compute fini
    te differencial operator
 * @param dir a int, direction in which we compute finite
    differencial operator
 * @param v a PnlVect pointer of nodal coefficients in dir
    ection dir
 * and hierarchic coefficients in others directions
 * @param G a GridSparse pointer
 * @param Ind_neig an int, index of left neighbour
 * @return a double equal to the discret finite difference operator
 */
double FD_Conv_Stencil_DeCenter(const int i,const int dir,
                                const PnlVect * v,
                                const GridSparse * G,
                                int Ind_neig)
{
  double res =GET(v,i)-GET(v,Ind_neig);
```

```
  res*=Domain_Size(G,dir)*
    (double)(1<<(Step(pnl_mat_int_get(G->Points,i,dir),
                     pnl_mat_int_get(G->Points,Ind_neig,
    dir))));
  return res;
}


/**
 * WARNING:Assumption homogen Grid,
 * Compute the second finite difference operator in direc
    tion dir, on point i
 *
 * @param i a int, index of point on which we compute fini
    te differencial operator
 * @param dir a int, direction in which we compute finite
    differencial operator
 * @param v a PnlVect pointer of nodal coefficients in dir
    ection dir
 * and hierarchic coefficients in others directions
 * @param G a GridSparse pointer
 * @return a double equal to the discret finite difference operator
 */
/*Assumption homogen Grid */
double FD_Lap_Center(const int i,const int dir,
                     const PnlVect * v,
                     GridSparse * G)
{
  int PP[3]={dir,i,0};
  int *Ind_neig=Ind_neig=pnl_hmat_int_lget(G->Ind_Neigh,PP)
    ;
  return FD_Lap_Stencil_Center(i,dir,v,G,*Ind_neig,*(Ind_ne
    ig+1));
}


/**
 * WARNING:Assumption homogen Grid,
 * Compute the first order finite difference operator with
    centered scheme in direction dir, on point i
 *
```

```
 * @param i a int, index of point on which we compute fini
    te differencial operator
 * @param dir a int, direction in which we compute finite
    differencial operator
 * @param v a PnlVect pointer of nodal coefficients in dir
    ection dir
 * and hierarchic coefficients in others directions
 * @param G a GridSparse pointer
 * @return a double equal to the discret finite difference operator
 */
/*Assumption homogen Grid */
double FD_Conv_Center(const int i,const int dir,
                      const PnlVect *v,
                      const GridSparse * G)
{
  int * Ind_neig;
  int PP[3]={dir,i,0};
  Ind_neig=pnl_hmat_int_lget(G->Ind_Neigh,PP);
  return FD_Conv_Stencil_Center(i,dir,v,G,*Ind_neig,*(Ind_
    neig+1));
}


/**
 * WARNING:Assumption homogen Grid,
 * Compute the first order finite difference operator with
    decentered scheme in direction dir, on point i
 * we use upwind scheme base of sign of coeff.
 *
 * @param i a int, index of point on which we compute fini
    te differencial operator
 * @param dir a int, direction in which we compute finite
    differencial operator
 * @param v a PnlVect pointer of nodal coefficients in dir
    ection dir
 * and hierarchic coefficients in others directions
 * @param G a GridSparse pointer
 * @param coeff a double, use to compute the good decentere
    d scheme
 * to have stabilisation properties on matrix operator
 * @return a double equal to the discret finite difference operator
```

```
 */
double FD_Conv_DeCenter(const int i,const int dir,
                        const PnlVect *v,
                        const GridSparse * G,
                        const double coeff)
{
  double Res;
  int Ind_neig;
  int PP[3]={dir,2,0};
  PP[2]=(coeff>0)?1:0;
  Ind_neig=pnl_hmat_int_get(G->Ind_Neigh,PP);
  Res=FD_Conv_Stencil_DeCenter(i,dir,v,G,Ind_neig);
  Res*=(coeff>0)?1:-1;
  return Res;
}


/*
 * Compute a function on a sparse grid
 *
 * @param G a GridSparse pointer
 * @param Vout a PnlVect pointer of nodal coefficients,
    equal to apply(x).
 * @param apply a function, the function interpolet on the
    sparse grid
 */
void GridSparse_apply_function(GridSparse * G, PnlVect *
    Vout, double (*apply)(const PnlVect *))
{
  int i;
  PnlVectInt * Current=pnl_vect_int_create(0);
  PnlVect *X=pnl_vect_create_from_zero(G->dim);
  for(i=1;i<G->size;i++)
    {
      pnl_mat_int_get_row(Current,G->Points,i);
      vect_int_dyadic_cast(Current,X);
      LET(Vout,i)=premia_pde_dim_boundary_eval_from_unit(
    apply,G->Bnd,X);
    }
  pnl_vect_int_free(&Current);
  pnl_vect_free(&X);
```

```
}

/*
 * Print function
 *
 * @param G a GridSparse pointer
 * @param Vout a PnlVect pointer of nodal coefficients,
 * @param file, the file where function is printing.
 */
void GridSparse_fprint(FILE * file,GridSparse * G,
                       PnlVect * Vout)
{
  int i=1,d=0;
  /* First point is no good points */
  PnlVect *X=pnl_vect_create_from_zero(G->dim);
  PnlVectInt * Current=pnl_vect_int_create(0);
  while(i<G->size)
    {
      pnl_mat_int_get_row(Current,G->Points,i);
      vect_int_dyadic_cast(Current,X);
      premia_pde_dim_boundary_from_unit_to_real_variable(G-
    >Bnd,X);
      d=0;
      while(d<X->size)
        {
          /*printf("%d,",d); */
          /*printf("%7.4f ",GET(X,d)); */
          fprintf(file,"%7.4f ",GET(X,d));
          d++;
        };
      /*printf("%7.4f{n",GET(Vout,i)); */
      fprintf(file,"%7.4f",GET(Vout,i));
      fprintf(file,"{n");
      i++;
    }
  pnl_vect_int_free(&Current);
  pnl_vect_free(&X);
}


/*//////////////////////////////////////////////////////////
```

```
    / */
/* Sparse Laplacien Operator */
/*//////////////////////////////////////////////////////////
    / */

/**
 * Create the sparse operator for Poisson equation
 *
 * @return pointer on LaplacienSparseOp
 */
LaplacienSparseOp * create_laplacien_sparse_operator()
{
  LaplacienSparseOp * Op = malloc(sizeof(LaplacienSparseOp)
    );
  return Op;
}

/**
 * initilisation of the sparse operator for Poisson equation
 *
 * @param Op pointer on LaplacienSparseOp
 */
void initialise_laplacien_sparse_operator(LaplacienSparseO
    p * Op)
{
  Op->V_tmp0=pnl_vect_create_from_zero(Op->G->size);
  Op->V_tmp1=pnl_vect_create_from_zero(Op->G->size);
}

/**
 * desaloccation of sparse operator for Poisson equation
 *
 * @param Op pointer on LaplacienSparseOp
 */
void laplacien_sparse_operator_free(LaplacienSparseOp **
    Op)
{
  pnl_vect_free(&(*Op)->V_tmp0);
  pnl_vect_free(&(*Op)->V_tmp1);
  GridSparse_free(&(*Op)->G);
  free(*Op);
```

```
  *Op=NULL;
}


/*
 * WARNING: with the convention A is matrix operator, we
    compute
 * V_out = a * A V_in + b Vout
 *
 * @param Op a LaplacienSparseOp contains data for abstract
     matrix-vector multiplication A Vin
 * @param Vin a PnlVec, input parameters
 * @param a a double
 * @param b a double
 * @param Vout a PnlVec, the output
 */
void GridSparse_apply_Laplacien(LaplacienSparseOp * Op,
    const PnlVect * Vin,
                                const double a,
                                const double b,
                                PnlVect * Vout)
/*Compute laplacien operator on Sparse Grid */
{
  int d,i;
  double scale=b;
  for(d=0;d<Op->G->dim;d++)
    {
      Hier_to_Nodal_in_dir(d,Vin,Op->V_tmp0,Op->G);
      /*pnl_vect_print(Op->V_tmp0); */
      for(i=1;i<Op->G->size;i++)
        LET(Op->V_tmp1,i)=FD_Lap_Center(i,d,Op->V_tmp0,Op->
    G);
      /*pnl_vect_print(Op->V_tmp1); */
      Nodal_to_Hier_in_dir(d,Op->V_tmp1,Op->V_tmp0,Op->G);
      /*pnl_vect_print(Op->V_tmp0); */
      /*pnl_vect_axpby(Vout,scale,Op->V_tmp0,a); */
      pnl_vect_axpby(a,Op->V_tmp0,scale,Vout);
      scale=1.0;
    }
}
```

```
/*
 * WARNING: with the convention A is matrix operator, we
    compute
 * V_out = a * PC V_in + b Vout
 *
 * @param Op a LaplacienSparseOp contains data for abstract
     matrix-vector multiplication PC Vin
 * @param Vin a PnlVec, input parameters
 * @param a a double
 * @param b a double
 * @param Vout a PnlVec, the output
 */
void GridSparse_apply_Laplacien_PC(LaplacienSparseOp * Op,
    const PnlVect * Vin,
                                   const double a,
                                   const double b,
                                   PnlVect * Vout)
{
  int i,d;
  double step_h,times_step_h;
  int Index[3]; /*Index[0]=dir, Index[1]= Position Index[2]
    = left or right */
  int *neig;
  pnl_vect_clone(Vout,Vin);
  Index[2]=0;
  i=1;
  do
    {
      times_step_h=1;
      for(d=0;d<Op->G->dim;d++)
        {
          Index[0]=d;
          Index[1]=i;
          neig  = pnl_hmat_int_lget(Op->G->Ind_Neigh,Index)
    ;
          step_h=(1<<(1*Step(pnl_mat_int_get(Op->G->Points,
    i,d),
           pnl_mat_int_get(Op->G->Points,*neig,d))));
          times_step_h*=step_h;
        }
      LET(Vout,i)/=1.0;/*sqrt(times_step_h); */
```

```
      /*
  times_step_h/
        (1-Op->theta_time_scheme*premia_pde_time_grid_step(
    Op->TG)*
  jacobi);
      */
      /*printf(" PC = %7.4f and Jacobi = %7.4f {n",GET(Op->
    PC,i),Op->theta_time_scheme*premia_pde_time_grid_step(Op->
    TG)*jacobi); */
      i++;
    }while(i<Op->G->size);

}

/*
 * Solve FD discret Sparse version of Poisson problem:
 * Laplacien Vout = Vin,
 *
 * @param Op a LaplacienSparseOp contains data for abstract
     matrix-vector multiplication PC Vin
 * @param Vin a PnlVec, the RHS
 * @param Vout a PnlVec, the output
 */
void GridSparse_Solve_Laplacien(LaplacienSparseOp * Op,
    const PnlVect * Vin,PnlVect * Vout)
{
  /*
    PnlGmresSolver* Solver;
    pnl_vect_set_double(Vout,0.0);
    Solver=pnl_gmres_solver_create(Vin->size,200,10,1e-6);
    pnl_gmres_solver_solve((void*)GridSparse_apply_Laplacie
    n,Op,(void*)GridSparse_apply_Laplacien_PC,Op,Vout,Vin,
    Solver);
    pnl_gmres_solver_free(&Solver);
  */
  PnlBicgSolver* Solver;
  Solver=pnl_bicg_solver_create(Vin->size,100,1e-6);
  pnl_bicg_solver_solve((void*)GridSparse_apply_Laplacien,
    Op,(void*)GridSparse_apply_Laplacien_PC,Op,Vout,Vin,Solver);
  pnl_bicg_solver_free(&Solver);
```

```
}

void GridSparse_Solve_Operator(GridSparse * G, const PnlVec
    t * Vin,PnlVect * Vout,
                               void (*operator)(GridSparse
    * , const PnlVect * , const double,const double, PnlVect *
    ),
                               void (*operator_PC)(
    GridSparse * , const PnlVect * , const double,const double, PnlVect
    *)
                               )
{
  PnlBicgSolver* Solver;
  pnl_vect_set_double(Vout,0.0);
  Solver=pnl_bicg_solver_create(Vin->size,200,1e-6);
  pnl_bicg_solver_solve((void*)operator,G,(void*)operator_
    PC,G,Vout,Vin,Solver);
  pnl_bicg_solver_free(&Solver);
}

/*//////////////////////////////////// */
/* Heat Operator on Sparse Grid */
/*//////////////////////////////////// */
/**
 * Create the sparse operator for heat equation
 *
 * @return pointer on HeatSparseOp
 */
HeatSparseOp * create_heat_sparse_operator(double eta)
{

  HeatSparseOp * Op = malloc(sizeof(HeatSparseOp));
  Op->eta=eta;
  Op->theta_time_scheme=1;
  return Op;
}

void GridSparse_preconditioner_heat_init (HeatSparseOp *
    Op)
{
  int i,d;
```

```
  double step_h,times_step_h;
  int Index[3]; /*Index[0]=dir, Index[1]= Position Index[2]
    = left or right */
  int *neig;
  Index[2]=0;
  i=1;
  do
    {
      times_step_h=0;
      for(d=0;d<Op->G->dim;d++)
        {
          Index[0]=d;
          Index[1]=i;
          neig  = pnl_hmat_int_lget(Op->G->Ind_Neigh,Index)
    ;
          step_h=Domain_Size(Op->G,d)*(1<<(Step(pnl_mat_
    int_get(Op->G->Points,i,d),
                                                    pnl_mat_
    int_get(Op->G->Points,*neig,d))));
          times_step_h+=step_h*step_h;
        }
      LET(Op->PC,i)=1.0;/*times_step_h; */
      /*
  times_step_h/
        (1-Op->theta_time_scheme*premia_pde_time_grid_step(
    Op->TG)*
  jacobi);
      */
      /*printf(" PC = %7.4f and Jacobi = %7.4f {n",GET(Op->
    PC,i),Op->theta_time_scheme*premia_pde_time_grid_step(Op->
    TG)*jacobi); */
      i++;
    }while(i<Op->G->size);
}

/**
 * initilisation of the sparse operator for heat equation
 *
 * @param Op pointer on HeatSparseOp
 */
void initialise_heat_sparse_operator(HeatSparseOp * Op)
```

```
{
  Op->V_tmp0=pnl_vect_create_from_zero(Op->G->size);
  Op->V_tmp1=pnl_vect_create_from_zero(Op->G->size);
  Op->PC=pnl_vect_create_from_zero(Op->G->size);
  GridSparse_preconditioner_heat_init (Op);
}

/**
 * desaloccation of sparse operator for heat equation
 *
 * @param Op pointer on HeatSparseOp
 */
void heat_sparse_operator_free(HeatSparseOp ** Op)
{
  premia_pde_time_grid_free(&(*Op)->TG);
  pnl_vect_free(&(*Op)->V_tmp0);
  pnl_vect_free(&(*Op)->V_tmp1);
  pnl_vect_free(&(*Op)->PC);
  GridSparse_free(&(*Op)->G);
  free(*Op);
  *Op=NULL;
}

/*
 * WARNING: with the convention A is matrix operator, we
    compute
 * V_out = a * A V_in + b Vout
 * Hera A = (Mass + sign Delta_t Laplacien_FD_Operator )
 *
 * @param Op a HeatSparseOp contains data for abstract matr
    ix-vector multiplication A Vin
 * @param Vin a PnlVec, input parameters
 * @param a a double
 * @param b a double
 * @param Vout a PnlVec, the output
 */
void GridSparse_apply_heat(HeatSparseOp * Op, const PnlVec
    t * Vin,
                           const double a,
                           const double b,
                           PnlVect * Vout)
```

```
{
  int d,i;
  double sg_delta=0.5*a*Op->eta*Op->eta*Op->theta_time_sche
    me*premia_pde_time_grid_step(Op->TG);
  /*>> Do V_out = a  Mass V_in + b Vout */
  /*pnl_vect_axpby(Vout,b,Vin,a); */
  pnl_vect_axpby(a,Vin,b,Vout);
  /*>> Do V_out += a  Op V_in  */
  for(d=0;d<Op->G->dim;d++)
    {
      Hier_to_Nodal_in_dir(d,Vin,Op->V_tmp0,Op->G);
      for(i=1;i<Op->G->size;i++)
        LET(Op->V_tmp1,i)=FD_Lap_Center(i,d,Op->V_tmp0,Op->
    G);
      Nodal_to_Hier_in_dir(d,Op->V_tmp1,Op->V_tmp0,Op->G);
      pnl_vect_axpby(sg_delta,Op->V_tmp0,1.0,Vout);
    }
}

/*
 * WARNING: with the convention A is matrix operator, we
    compute
 * V_out = a * PC V_in + b Vout
 *
 * @param Op a HeatSparseOp contains data for abstract matr
    ix-vector multiplication PC Vin
 * @param Vin a PnlVec, input parameters
 * @param a a double
 * @param b a double
 * @param Vout a PnlVec, the output
 */
void GridSparse_apply_heat_PC(HeatSparseOp * Op, const PnlV
    ect * Vin,
                              const double a,
                              const double b,
                              PnlVect * Vout)
{ pnl_vect_axpby(1.0,Vin,0.0,Vout);
  pnl_vect_mult_vect_term(Vout,Op->PC);
}

/*
```

```
 * Solve FD discret Sparse version of Heat problem with th
    eta-scheme in time
 * Vout - theta  delta_t  1/2 Laplacien_Operator  Vout
 * = Vin + (1- theta) delta_t  1/2 Laplacien_Operator  Vin,

 *
 * @param Op a HeatSparseOp contains data for abstract matr
    ix-vector multiplication PC Vin
 * @param Vin a PnlVec, the RHS
 * @param Vout a PnlVec, the output
 */
void GridSparse_Solve_heat(HeatSparseOp * Op, const PnlVec
    t * Vin,PnlVect * Vout)
{

  PnlVect* V_rhs;
  PnlBicgSolver* Solver;
  double theta=0.0;
  V_rhs=pnl_vect_create_from_zero(Vin->size);
  pnl_vect_clone(Vout,Vin);
  Solver=pnl_bicg_solver_create(Vin->size,200,1e-6);
  premia_pde_time_start(Op->TG);
  do
    {
      Op->theta_time_scheme=theta;
      GridSparse_apply_heat(Op,Vout,1.0,0.0,V_rhs);
      Op->theta_time_scheme=-1.0+theta;
      pnl_bicg_solver_solve((void*)GridSparse_apply_heat,
                            Op,(void*)
                            GridSparse_apply_heat_PC,
                            Op,Vout,V_rhs,Solver);
    }while(premia_pde_time_grid_increase(Op->TG));
  pnl_bicg_solver_free(&Solver);
  pnl_vect_free(&V_rhs);
}

#undef SPARSE_N2H
#undef SPARSE_H2N
#undef SPARSE_N2H_FUNC
```

References