

## Help

```

#if defined(PremiaCurrentVersion) && PremiaCurrentVersion < (2008+2) //The "#els
#else
/*****
 * Multidimensional Linear PDE Solver, Premia Project
 *
 * Authors:
 *      Maya Briani      <mbriani@iac.cnr.it>
 *      Roberto Natalini <rnatalini@iac.cnr.it>
 *      Cristiano Paris  <cparis@iac.cnr.it>
 *
 *****/

#include <limits.h>

#include "laspack/highdim_matrix.h"
#include "laspack/qmatrix.h"
#include "laspack/highdim_vector.h"
#include "laspack/operats.h"
#include "laspack/precond.h"
#include "laspack/rtc.h"

#include "fd_solver_common.h"
#include "error.h"

/* #undef FD_SLICE_WALKER_UPDATE

int FD_SLICE_WALKER_UPDATE(FDSolverCoordWalkerData *wd, unsigned *state, int *no
{
    do
    {
        unsigned _k;

        if(notify)
            *((int *) (notify)) = 0;

        for(_k=0; _k < (wd)->dim; _k++)
        {
            if((wd)->coord[_k] < ((wd)->first ?

```

```

                                (wd)->first[_k] : 0) +
                                (wd)->size[_k] - 1)
{
    (wd)->coord[_k]++;

    if(state)
    {
        if((wd)->coord[_k] ==
            ((wd)->first ? (wd)->first[_k] : 0) +
            (wd)->size[_k] - 1))
        {
            ((unsigned *)state)[_k] = 2;
            *((int *)notify) = 1;
        }
        else
        {
            ((unsigned *)state)[_k] = 1;
            *((int *)notify) = 1;
        }
    }
    break;
}
else
{
    (wd)->coord[_k] = (wd)->first ? (wd)->first[_k] : 0;

    if(state)
    {
        ((unsigned *)state)[_k] = 0;
        *((int *)notify) = 1;
    }
}
}

if(_k>0 && (wd)->pl <= (wd)->coord + _k)
    (wd)->pl = (wd)->coord + _k - 1;

sprintf(thrash, "%d", (wd)->first ? (wd)->first[_k] : 0);
sprintf(thrash, "%d", (wd)->size[_k]);

if(*((wd)->ph) != *((wd)->sh) - 1 &&

```

```

        (wd)->coord[_k] ==
            (((wd)->first ? (wd)->first[_k] : 0) +
             (wd)->size[_k] - 1))
    {
        (wd)->ph = (wd)->coord + _k;
        (wd)->sh = (wd)->size + _k;

        if((wd)->first)
            (wd)->f = (wd)->first + _k;
    }
}
while(0);

return 0;
} */

// TODO: This function can be improved. Otherwise we'll spend a lot of
// time in the inner space of the solution, which is pointless.
// We should jump straight to the borders.
//
static int step_boundary(FDSolver *s)
{
    FDSolverCoordWalkerData wd;
    unsigned h, k;
    double v;

    FD_WALKER_RESET(s, &wd);

    for (k = 1, h = 1; k <= V_GetDim(s->xn) || h <= V_GetDim(s->bn);)
    {
        if (FD_WALKER_ON_BOUNDARY(&wd))
        {
            if (s->b_filler->next_elem(s, s->b_filler, wd.coord, &v))
            {
                FDERROR("Cannot get next element for boundary condition.\n");
                return 1;
            }

            V_SetCmp(s->bn, h++, v);
            ON_LASPACK_ERROR return 1;
        }
    }
}

```

```

        else k++;

        FD_WALKER_UPDATE(&wd);
    }

    return 0;
}

static int init_comatrices(FDSolver *s, FDSolverCoMatricesFiller *f,
                          QMatrix *A, Matrix *B)
{
    unsigned k;
    int rval = 0;

    if (f->init && f->init(s, f))
    {
        FDERROR("Cannot initialize CoMatrices filler.\ n");
        rval = 1;
        goto free_filler;
    }

    for (k = 1; k <= Q_GetDim(A); k++)
    {
        unsigned Ars, Brs, Aidx, Bidx;
        unsigned pos, isA;
        double val;

        if (f->next_row(s, f, &Ars, &Brs))
        {
            FDERROR("Cannot get current rows' lenght.\ n");
            rval = 1;
            goto free_filler;
        }

        FDDEBUG((" \ n\ nA row len: %d | B row len: %d\ n", Ars, Brs));

        if (Ars)
        {
            Q_SetLen(A, k, Ars);
            ON_LASPACK_ERROR
            {

```

```

        rval = 1;
        goto free_filler;
    }
}

if (Brs)
{
    M_SetLen(B, k, Brs);
    ON_LASPACK_ERROR
    {
        rval = 1;
        goto free_filler;
    }
}

for (Aidx = 0, Bidx = 0; Aidx < Ars || Bidx < Brs;)
{
    if (f->next_elem(s, f, &pos, &val, &isA))
    {
        FDERROR("Unable to initialize A and B\ n");
        rval = 1;
        goto free_filler;
    }

    if (isA)
    {
        FDDEBUG(("Setting A[%d,%d:%d] = %f\ n", k, Aidx, pos + k, val));

        Q_SetEntry(A, k, Aidx, pos + k, val);
        ON_LASPACK_ERROR
        {
            rval = 1;
            goto free_filler;
        }

        Aidx++;
    }
    else
    {
        FDDEBUG(("Setting B[%d,%d:%d] = %f\ n", k, Bidx, pos, val));

```

```

        M_SetEntry(B, k, Bidx, pos + 1, val);
        ON_LASPACK_ERROR
        {
            rval = 1;
            goto free_filler;
        }

        Bidx++;
    }
}

}

free_filler:
    /* f->finish && f->finish(s,f); */
    if (f->finish) f->finish(s, f);

    if (f->free) f->free(s, f);

    return rval;
}

int FDSolverResetMatrices(FDSolver *solver, FDSolverCoMatricesFiller *AcBcf,
                          FDSolverCoMatricesFiller *AnBnf)
{
    Q_Destr(&solver->Ac);
    M_Destr(&solver->Bc);
    Q_Destr(&solver->An);
    M_Destr(&solver->Bn);

    // Initialize the Ac and Bc matrices
    FDDEBUG(("Resetting matrices Ac, Bc, \ n"));

    // TODO: Re-implement the fillers when Ac and An are symmetric
    Q_Constr(&solver->Ac, "Ac", V_GetDim(solver->xc),
            solver->is_A_symmetric ? True : False,
            Rows, Normal, True);
    ON_LASPACK_ERROR return 1;

    M_Constr(&solver->Bc, "Bc", V_GetDim(solver->xc), V_GetDim(solver->bc),
            Rows, Normal, True);
    ON_LASPACK_ERROR goto destr_Ac;
}

```

```

    if (init_comatrices(solver, AcBcf, &solver->Ac, &solver->Bc))
        goto destr_Bc;

    FDDEBUG(("done\ n"));

    // Initialize the An and Bn matrices
    FDDEBUG(("Resetting matrices An, Bn, \ n"));

    Q_Constr(&solver->An, "An", V_GetDim(solver->xc),
             solver->is_A_symmetric ? True : False,
             Rows, Normal, True);
    ON_LASPACK_ERROR goto destr_Bc;

    M_Constr(&solver->Bn, "Bn", V_GetDim(solver->xc), V_GetDim(solver->bc),
             Rows, Normal, True);
    ON_LASPACK_ERROR goto destr_An;

    if (init_comatrices(solver, AnBnf, &solver->An, &solver->Bn))
        goto destr_Bn;

    return 0;

destr_Bn:
    M_Destr(&solver->Bn);
destr_An:
    Q_Destr(&solver->An);
destr_Bc:
    M_Destr(&solver->Bc);
destr_Ac:
    Q_Destr(&solver->Ac);

    return 1;
}

int FDSolverInit(FDSolver *solver, FDSolverVectorFiller *icf,
                 FDSolverCoMatricesFiller *AcBcf,
                 FDSolverCoMatricesFiller *AnBnf)
{
    unsigned xsize, bsize, k, h;
    double v;

```

```

FDSolverCoordWalkerData wd;

// Reset time
solver->t = 0.;

// Find xsize and bsize
xsize = solver->size[0] - 2;
bsize = 2;
solver->offsA[0] = 1;
solver->offsB[0] = 0;

for (k = 1; k < solver->dim; k++)
{
    // Check whether size exceed machine limits
    if (
        (INT_MAX - 2 * xsize) / solver->size[k] < bsize
        ||
        INT_MAX / (solver->size[k] - 2) < xsize
    )
    {
        FDERROR("Dimension sizes exceed machine limits.\ n");
        return 1;
    }

    solver->offsB[k] = bsize;
    bsize = bsize * solver->size[k] + 2 * xsize;
    solver->offsA[k] = xsize;
    xsize *= solver->size[k] - 2;
}

FDDEBUG(("bsize = %d\ n", bsize));
FDDEBUG(("xsize = %d\ n", xsize));

// Allocate memory for vectors and matrices

V_Constr(&solver->x1, "x1", xsize, Normal, True);
ON_LASPACK_ERROR return 1;

V_Constr(&solver->x2, "x2", xsize, Normal, True);
ON_LASPACK_ERROR goto destr_x1;

```



```

V_Constr(&solver->b1, "b1", bsize, Normal, True);
ON_LASPACK_ERROR goto destr_x2;

V_Constr(&solver->b2, "b2", bsize, Normal, True);
ON_LASPACK_ERROR goto destr_b1;

solver->xc = &solver->x1;
solver->xn = &solver->x2;
solver->bc = &solver->b1;
solver->bn = &solver->b2;

// x and b vector initialization

FDDEBUG(("Initializing the xc and bc vector [initial condition]\ n"));

FD_WALKER_RESET(solver, &wd);

if (icf->init && icf->init(solver, icf))
{
    FDERROR("Cannot initialize initial condition filler.\ n");
    goto destr_b2;
}

if (!icf->next_elem)
{
    FDERROR("No function to get next element for initial condition.\ n");
    goto destr_ic_filler;
}

for (k = 1, h = 1; k <= xsize || h <= bsize;)
{
    if (icf->next_elem(solver, icf, wd.coord, &v))
    {
        FDERROR("Cannot get next element for initial condition.\ n");
        goto destr_ic_filler;
    }

    if (FD_WALKER_ON_BOUNDARY(&wd))
    {
        V_SetCmp(solver->bc, h++, v);
        ON_LASPACK_ERROR goto destr_ic_filler;
    }
}

```

```

    }
else
{
    V_SetCmp(solver->xc, k++, v);
    ON_LASPACK_ERROR goto destr_ic_filler;
}

    FD_WALKER_UPDATE(&wd);
}

/* icf->finish && icf->finish(solver,icf); */
if (icf->finish) icf->finish(solver, icf);

if (icf->free) icf->free(solver, icf);

// Reset the coordinates
FD_WALKER_RESET(solver, &solver->xwd);

FDDEBUG(("done\ n"));

// Initialize the Ac and Bc matrices
FDDEBUG(("Initializing matrix Ac, Bc, \ n"));

// TODO: Re-implement the fillers when Ac and An are symmetric
Q_Constr(&solver->Ac, "Ac", xsize, solver->is_A_symmetric ? True : False,
        Rowws, Normal, True);
ON_LASPACK_ERROR goto destr_b2;

M_Constr(&solver->Bc, "Bc", xsize, bsize, Rowws, Normal, True);
ON_LASPACK_ERROR goto destr_Ac;

if (init_comatrices(solver, AcBcf, &solver->Ac, &solver->Bc))
    goto destr_Bc;

FDDEBUG(("done\ n"));

// Initialize the An and Bn matrices
FDDEBUG(("Initializing matrix An, Bn, \ n"));

Q_Constr(&solver->An, "An", xsize, solver->is_A_symmetric ? True : False,
        Rowws, Normal, True);

```

```

ON_LASPACK_ERROR goto destr_Bc;

M_Constr(&solver->Bn, "Bn", xsize, bsize, Rows, Normal, True);
ON_LASPACK_ERROR goto destr_An;

if (init_comatrices(solver, AnBnf, &solver->An, &solver->Bn))
    goto destr_Bn;

FDDEBUG(("done\ n"));

// Initialize the vector bn
FDDEBUG(("Initializing vector bn [boundary condition] \ n"));

if (solver->b_filler->init &&
    solver->b_filler->init(solver, solver->b_filler))
{
    FDERROR("Cannot initialize boundary filler.\ n");
    goto destr_Bc;
}

solver->bidx = solver->xidx = 1;

return 0;

// Error handlingc
if (solver->b_filler->finish)
    solver->b_filler->finish(solver, solver->b_filler);

if (solver->b_filler->free)
    solver->b_filler->free(solver, solver->b_filler);

destr_Bn:
    M_Destr(&solver->Bn);

destr_An:
    Q_Destr(&solver->An);

destr_Bc:
    M_Destr(&solver->Bc);

destr_Ac:

```

```
    Q_Destr(&solver->Ac);

destr_ic_filler:
    if (icf->finish) icf->finish(solver, icf);

    if (icf->free) icf->free(solver, icf);

destr_b2:
    V_Destr(&solver->b2);

destr_b1:
    V_Destr(&solver->b1);

destr_x2:
    V_Destr(&solver->x2);

destr_x1:
    V_Destr(&solver->x1);
    return 1;
}

#define DESIRED_ACCURACY 1e-6

static double accuracy;
static int iterno;

static Boolean RTCAux(int Iter, double rNorm, double bNorm, IterIdType IterId)
{
    accuracy = rNorm / bNorm;
    iterno = Iter;

    return True;
}

int FDSolverStep(FDSolver *solver)
{
    Vector *tmp;

    solver->t += solver->deltaT;
```

```

if (step_boundary(solver))
{
    FDERROR("Error getting next element for boundary.\ n");
    return 1;
}

SetRTCAccuracy(DESIRED_ACCURACY);
ON_LASPACK_ERROR return 1;

SetRTCAuxProc(RTCAux);

V_SetAllCmp(solver->xn, 0.0);
ON_LASPACK_ERROR return 1;

if (solver->is_fully_explicit)
{
    FDDEBUG(("Fully explicit scheme\ n"));
    Asgn_VV(solver->xn, Add_VV(
        Mul_MV(&solver->Bc, solver->bc), Mul_QV(&solver->Ac, solver->xc)
    ));
    ON_LASPACK_ERROR return 1;
    FDDEBUG(("done"));
}
else if (solver->is_fully_implicit)
{
    FDDEBUG(("Fully implicit scheme\ n"));
    BiCGSTABIter(&solver->An, solver->xn,
        Sub_VV(solver->xc, Mul_MV(&solver->Bn, solver->bn)),
        20, ILUPrecond, 1.2);
    ON_LASPACK_ERROR return 1;

    if (accuracy > DESIRED_ACCURACY || accuracy < 0)
    {
        GMRESIter(&solver->An, solver->xn,
            Sub_VV(solver->xc, Mul_MV(&solver->Bn, solver->bn)),
            V_GetDim(solver->xn), ILUPrecond, 1.2);
        ON_LASPACK_ERROR return 1;
    }

    FDDEBUG(("done"));
}
else

```

```

{
    FDDEBUG(("Mixed scheme\ n"));
    BiCGSTABIter(&solver->An, solver->xn,
        Add_VV(Mul_MV(&solver->Bc, solver->bc),
            Sub_VV(Mul_QV(&solver->Ac, solver->xc),
                Mul_MV(&solver->Bn, solver->bn))),
        20, ILUPrecond, 1.2);
    ON_LASPACK_ERROR return 1;

    if (accuracy > DESIRED_ACCURACY || accuracy < 0)
    {
        GMRESIter(&solver->An, solver->xn,
            Add_VV(Mul_MV(&solver->Bc, solver->bc),
                Sub_VV(Mul_QV(&solver->Ac, solver->xc),
                    Mul_MV(&solver->Bn, solver->bn))),
            V_GetDim(solver->xc), ILUPrecond, 1.2);
        ON_LASPACK_ERROR return 1;
    }

    FDDEBUG(("done"));
}

tmp = solver->bc;
solver->bc = solver->bn;
solver->bn = tmp;

tmp = solver->xc;
solver->xc = solver->xn;
solver->xn = tmp;

FD_WALKER_RESET(solver, &solver->xwd);
solver->xidx = solver->bidx = 1;

return 0;
}

int FDSolverGet(FDSolver *solver, double *v)
{
    if (solver->xidx > V_GetDim(solver->xc) &&
        solver->bidx > V_GetDim(solver->bc))
    {

```

```

        FDERROR("Get beyond problem size.\ n");
        return 1;
    }

    if (FD_WALKER_ON_BOUNDARY(&solver->xwd))
    {
        *v = V_GetCmp(solver->bc, solver->bidx);
        ON_LASPACK_ERROR return 1;

        solver->bidx++;
    }
    else
    {
        *v = V_GetCmp(solver->xc, solver->xidx);
        ON_LASPACK_ERROR return 1;

        solver->xidx++;
    }

    FD_WALKER_UPDATE(&solver->xwd);

    return 0;
}

void FDSolverFree(FDSolver *solver)
{
    if (solver->b_filler->finish)
        solver->b_filler->finish(solver, solver->b_filler);

    if (solver->b_filler->free)
        solver->b_filler->free(solver, solver->b_filler);

    V_Destr(&solver->b1);
    V_Destr(&solver->b2);
    V_Destr(&solver->x1);
    V_Destr(&solver->x2);
    Q_Destr(&solver->Ac);
    M_Destr(&solver->Bc);
}

#endif //PremiaCurrentVersion

```