

Help

```

#if defined(PremiaCurrentVersion) && PremiaCurrentVersion < (2008+2) //The "#els
#else

/*****
*   CPS - A simple C PDE solver                               *
*                                                                 *
*   Copyright (c) 2007,                                       *
*   Maya Briani      <m.briani@iac.rm.cnr.it>,               *
*   Francesco Ferreri <francesco.ferreri@gmail.com>,         *
*   Roberto Natalini <r.natalini@iac.rm.cnr.it>,             *
*   Marco Papi       <m.papi@iac.rm.cnr.it>                  *
*                                                                 *
*****/
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "cps_types.h"
#include "cps_grid.h"
#include "cps_grid_tuner.h"
#include "cps_utils.h"
#include "cps_debug.h"
#include "cps_stencil.h" /* WARNING: this may be a cohesion/coupling error ..
#include "cps_assertions.h"

/* helpful macros */

#define TICK_TO_VALUE(g,t,d) \
    (g)->min_value[d] + ((double)t * (g)->delta[d])

#define UPDATE_CURRENT_VALUE(g,d) \
    CHECK("valid_dimension",((d) <= (g)->space_dimensions && (d) >= 0)); \
    (g)->current_value[d] = (g)->min_value[d] + ((double)(g)->current_tick[d] * (g)

#define UPDATE_CURRENT_VALUES(g) \
    {int m_dim; \
    for(m_dim = T_DIM; m_dim <= (g)->space_dimensions; m_dim++){ \
        UPDATE_CURRENT_VALUE(g,m_dim); \
    }}

```

```

#define CORE_AFTER_DIM(g,d) \
    ((g)->current_tick[(d)] == ((g)->ticks[(d)] - 1))

#define PLAIN_AFTER_DIM(g,d) \
    ((g)->current_tick[(d)] == ((g)->ticks[(d)]))

/* private implementation functions */
static int node_lexicographic_order(const grid_node *node)
{
    int result = 0;
    const grid *grid;

    /* compute lexicographic order for given node */
    REQUIRE("node_not_null", node != NULL);
    REQUIRE("grid_is_set", node->source_grid != NULL);
    REQUIRE("grid_node_is_internal", grid_node_is_internal(node));

    grid = node->source_grid;
    /* TODO: generalize to n-dimension case !!! */

    result = (!grid_iterator_first(grid, X_DIM)) +
        node->tick[X_DIM] +
        (node->tick[Y_DIM] - (grid_iterator_first(grid, Y_DIM))) * (grid_iter

    ENSURE("valid_result", result > 0);
    return result;
}

static int position_shift(int pos, int x, int y, int *sx, int *sy)
{
    /* compute dimensions shift according to position */
    REQUIRE("valid_position", pos >= XY && pos <= XPYP);

    switch (pos)
    {
        case XY:
            *sx = x;
            *sy = y;
            break;
    }
}

```

```
    case XPY:
        *sx = x + 1;
        *sy = y;
        break;
    case XPYM:
        *sx = x + 1;
        *sy = y - 1;
        break;
    case XYM:
        *sx = x;
        *sy = y - 1;
        break;
    case XMYM:
        *sx = x - 1;
        *sy = y - 1;
        break;
    case XMY:
        *sx = x - 1;
        *sy = y;
        break;
    case XMYP:
        *sx = x - 1;
        *sy = y + 1;
        break;
    case XYP:
        *sx = x;
        *sy = y + 1;
        break;
    case XPYP:
        *sx = x + 1;
        *sy = y + 1;
        break;
}
return OK;
}

static int space_forth(grid *g, int dim)
{
    /* recursively incremente current_ticks */
    REQUIRE("grid_not_null", g != NULL);
    REQUIRE("grid_is_rescaled", g->is_rescaled);
```

```

    REQUIRE("valid_dimension", dim > T_DIM && dim <= g->space_dimensions);

    g->current_tick[dim]++;
    if (g->current_tick[dim] > grid_iterator_last(g, dim))
    {
        if (dim < g->space_dimensions)
        {
            if (space_forth(g, dim + 1))
            {
                g->current_tick[dim] = grid_iterator_first(g, dim);
                return 1;
            }
        }
        return 0;
    }
    return 1;
}

/*
 * public interface functions
 */

int grid_create(grid **g)
{
    STANDARD_CREATE(g, grid);
    return OK;
}

int grid_destroy(grid **g)
{
    /* destroy grid and tuner */
    if ((*g)->tuner)
        grid_tuner_destroy(&((*g)->tuner));

    STANDARD_DESTROY(g);
    return OK;
}

int grid_rescale(grid *g)
{
    /* rescale grid around focus */

```

```

    REQUIRE("grid_not_null", g != NULL);
    REQUIRE("focus_is_set", g->focus[X_DIM] != 0 && g->focus[Y_DIM] != 0);
    REQUIRE("rescale_tuner_set", g->tuner->tuners[RESCALE_TUNER] != NULL);

    grid_tuner_apply(g->tuner, RESCALE_TUNER, g);
    g->is_rescaled = 1;

    ENSURE("grid_is_rescaled", g->is_rescaled);
    return OK;
}

int grid_set_tuner(grid *g, grid_tuner *tuner)
{
    /* set tuner */
    REQUIRE("grid_not_null", g != NULL);
    REQUIRE("tuner_not_null", tuner != NULL);

    g->tuner = tuner;

    ENSURE("tuner_set", g->tuner == tuner);
    return OK;
}

int grid_set_focus(grid *g, int dim, double value)
{
    /* set focus of grid for dimension 'dim' */
    REQUIRE("grid_not_null", g != NULL);
    REQUIRE("valid_dimension", dim >= 0 && dim <= g->space_dimensions);

    g->focus[dim] = value;
    g->is_rescaled = 0;

    ENSURE("not_is_rescaled", !g->is_rescaled);
    return OK;
}

int grid_set_space_dimensions(grid *g, int dims)
{
    /* set number of space dimensions (apart from time)*/
    REQUIRE("grid_not_null", g != NULL);
    REQUIRE("valid_number_of_dimensions", dims > 0 && dims < MAX_DIMENSIONS);

```

```
g->space_dimensions = dims;

ENSURE("dimensions_set", g->space_dimensions == dims);
return OK;
}

int grid_set_min_value(grid *g, int dim, double value)
{
    /* set min (left) value for given dimension */
    REQUIRE("grid_not_null", g != NULL);
    REQUIRE("valid_dimension", dim >= 0 && dim <= g->space_dimensions);

    g->min_value[dim] = value;
    g->is_rescaled = 0;

    ENSURE("grid_is_not_tuned", !g->is_rescaled);
    return OK;
}

int grid_set_max_value(grid *g, int dim, double value)
{
    /* set max (right) value for given dimension */

    REQUIRE("grid_not_null", g != NULL);
    REQUIRE("valid_dimension", dim >= 0 && dim <= g->space_dimensions);
    REQUIRE("really_a_maximum", value > g->min_value[dim]);

    g->max_value[dim] = value;
    g->is_rescaled = 0;

    ENSURE("grid_is_not_tuned", !g->is_rescaled);
    return OK;
}

int grid_set_ticks(grid *g, int dim, int ticks)
{
    /* set number of ticks for given dimension */

    REQUIRE("grid_not_null", g != NULL);
    REQUIRE("valid_dimension", dim >= 0 && dim <= g->space_dimensions);
```

```

    REQUIRE("ticks_gt_one", ticks > 1);

    g->ticks[dim] = ticks;
    g->is_rescaled = 0;

    ENSURE("grid_is_not_tuned", !g->is_rescaled);
    ENSURE("ticks_set", g->ticks[dim] == ticks);
    return OK;
}

int grid_set_iterator(grid *g, int dim, int type)
{
    /* sets iterator type for given space dimension */
    REQUIRE("grid_not_null", g != NULL);
    REQUIRE("valid_dimension", dim > T_DIM && dim <= g->space_dimensions);
    REQUIRE("valid_iterator_type", type == ITER_PLAIN || type == ITER_CORE);

    g->current_iterator[dim] = type;

    ENSURE("iterator_set", g->current_iterator[dim] == type);
    return OK;
}

int grid_set_all_iterators(grid *g, int type)
{
    int dim;
    /* sets iterator to type along all space dimensions */
    REQUIRE("grid_not_null", g != NULL);
    REQUIRE("valid_iterator_type", type == ITER_PLAIN || type == ITER_CORE);

    for (dim = X_DIM; dim <= g->space_dimensions; dim++)
    {
        g->current_iterator[dim] = type;
    }

    return OK;
}

int grid_time_initial(grid *g)

```

```
{
    /* put time iterator to start (t==0) */

    REQUIRE("grid_not_null", g != NULL);
    REQUIRE("grid_is_tuned", g->is_tuned);

    g->current_tick[T_DIM] = 0;
    UPDATE_CURRENT_VALUE(g, T_DIM);

    return OK;
}

int grid_time_start(grid *g)
{
    /* put time iterator to start (t=1) */

    REQUIRE("grid_not_null", g != NULL);
    REQUIRE("grid_is_tuned", g->is_tuned);

    g->current_tick[T_DIM] = 1;
    UPDATE_CURRENT_VALUE(g, T_DIM);

    return OK;
}

int grid_time_forth(grid *g)
{
    /* step one tick forth in time */

    REQUIRE("g_not_null", g != NULL);
    REQUIRE("grid_is_tuned", g->is_tuned);
    REQUIRE("not_time_after", !grid_time_after(g));

    g->current_tick[T_DIM]++;
    UPDATE_CURRENT_VALUE(g, T_DIM);

    return OK;
}

int grid_time_after(const grid *g)
{

```



```

/* true when grid time is at end */

    REQUIRE("g_not_null", g != NULL);
    REQUIRE("grid_is_tuned", g->is_tuned);

    return (g->current_tick[T_DIM] == (g->ticks[T_DIM])
        || (g->current_value[T_DIM] > g->max_value[T_DIM]));
}

/* plain iterators */

int grid_iterator_span(const grid *grid, int dim)
{
    int result = 0;
    /* returns span of iterator currently set for iteration */
    REQUIRE("grid_not_null", grid != NULL);
    REQUIRE("valid_dimension", dim >= X_DIM && dim <= grid->space_dimensions);

    switch (grid->current_iterator[dim])
    {
        case ITER_CORE:
            result = grid->ticks[dim] - 2;
            break;
        case ITER_PLAIN:
            result = grid->ticks[dim];
            break;
    }
    return result;
}

int grid_iterator_first(const grid *grid, int dim)
{
    int result = 0;
    /* first tick for given dim according to iterator type */
    REQUIRE("grid_not_null", grid != NULL);
    REQUIRE("valid_dimension", dim >= X_DIM && dim <= grid->space_dimensions);

```

```

switch (grid->current_iterator[dim])
{
    case ITER_CORE:
        result = 1;
        break;
    case ITER_PLAIN:
        result = 0;
        break;
}

ENSURE("result_is_zero_xor_one", result == 0 || result == 1);
return result;
}

int grid_iterator_last(const grid *grid, int dim)
{
    int result = 0;
    /* last tick for given dim according to iterator type */
    REQUIRE("grid_not_null", grid != NULL);
    REQUIRE("valid_dimension", dim >= X_DIM && dim <= grid->space_dimensions);

    switch (grid->current_iterator[dim])
    {
        case ITER_CORE:
            result = grid->ticks[dim] - 2;
            break;
        case ITER_PLAIN:
            result = grid->ticks[dim] - 1;
            break;
    }
    return result;
}

int grid_plain_start(grid *g, int dim)
{
    /* start grid in plain mode */
    REQUIRE("grid_not_null", g != NULL);

```

```

    REQUIRE("grid_is_rescaled", g->is_rescaled);
    REQUIRE("valid_dimension", dim > T_DIM && dim <= g->space_dimensions);

    g->current_tick[dim] = 0;
    UPDATE_CURRENT_VALUE(g, dim);

    return OK;
}

int grid_plain_forth(grid *g, int dim)
{
    REQUIRE("grid_not_null", g != NULL);
    REQUIRE("grid_is_rescaled", g->is_rescaled);
    REQUIRE("valid_dimension", dim > T_DIM && dim <= g->space_dimensions);
    REQUIRE("not_after", !grid_plain_after(g, dim));

    g->current_tick[dim]++;
    UPDATE_CURRENT_VALUE(g, dim);

    return OK;
}

int grid_plain_after(const grid *g, int dim)
{
    /* true when iterator is after last node over dim */
    REQUIRE("grid_not_null", g != NULL);
    REQUIRE("grid_is_rescaled", g->is_rescaled);
    REQUIRE("valid_dimension", dim > T_DIM && dim <= g->space_dimensions);

    return (g->current_tick[dim] == g->ticks[dim]);
}

/* core iterators */

int grid_core_start(grid *g, int dim)
{
    /* start grid in core mode for given dimension */
    REQUIRE("grid_not_null", g != NULL);
    REQUIRE("grid_is_rescaled", g->is_rescaled);
    REQUIRE("valid_dimension", dim > T_DIM && dim <= g->space_dimensions);

```

```

    g->current_tick[dim] = 1;
    UPDATE_CURRENT_VALUE(g, dim);

    return OK;
}

int grid_core_forth(grid *g, int dim)
{
    REQUIRE("grid_not_null", g != NULL);
    REQUIRE("grid_is_rescaled", g->is_rescaled);
    REQUIRE("valid_dimension", dim > T_DIM && dim <= g->space_dimensions);
    REQUIRE("not_after", !grid_core_after(g, dim));

    g->current_tick[dim]++;
    UPDATE_CURRENT_VALUE(g, dim);

    return OK;
}

int grid_core_after(const grid *g, int dim)
{
    /* true when iterator is after last node over dim */
    REQUIRE("grid_not_null", g != NULL);
    REQUIRE("grid_is_rescaled", g->is_rescaled);
    REQUIRE("valid_dimension", dim > T_DIM && dim <= g->space_dimensions);

    return (g->current_tick[dim] == (g->ticks[dim] - 1));
}

/* non-dimensional iterators */

int grid_space_start(grid *g)
{
    int dim;
    /* set iterator at start (node 1,1,...: first
       non-boundary node) */
    REQUIRE("grid_not_null", g != NULL);
    REQUIRE("grid_is_rescaled", g->is_rescaled);

```

```

    for (dim = X_DIM; dim <= g->space_dimensions; dim++)
    {
        g->current_tick[dim] = grid_iterator_first(g, dim);
        UPDATE_CURRENT_VALUE(g, dim);
    }
    return OK;
}

int grid_space_forth(grid *g)
{
    /* step forth iterator */
    REQUIRE("grid_not_null", g != NULL);
    REQUIRE("grid_is_rescaled", g->is_rescaled);
    REQUIRE("iterator_not_after", !grid_space_after(g));

    space_forth(g, X_DIM);
    UPDATE_CURRENT_VALUES(g);

    return OK;
}

int grid_space_after(const grid *g)
{
    int dim;
    int result = 1;

    /* true when iterator is after
       last non-boundary node */
    REQUIRE("grid_not_null", g != NULL);
    REQUIRE("grid_is_rescaled", g->is_rescaled);

    for (dim = X_DIM; dim <= g->space_dimensions; dim++)
    {
        result = result && (g->current_tick[dim] > grid_iterator_last(g, dim));
    }
    return result;
}

/* guard iterators */

```

```
int grid_guard_start(grid *grid)
{
    int dim;
    /* start iteration over guard nodes */
    REQUIRE("grid_not_null", grid != NULL);

    for (dim = X_DIM; dim <= grid->space_dimensions; dim++)
    {
        grid->current_tick[dim] = grid_iterator_first(grid, dim);
    }
    UPDATE_CURRENT_VALUES(grid);
    return OK;
}

int grid_guard_forth(grid *grid)
{
    int x, y;
    /* goes forth to next guard node */
    REQUIRE("grid_not_null", grid != NULL);
    REQUIRE("not_after", !grid_guard_after(grid));

    x = grid->current_tick[X_DIM];
    y = grid->current_tick[Y_DIM];

    /* TODO: refactor for n-dimensions */
    if (y == grid_iterator_first(grid, Y_DIM))
    {
        if (x < (grid_iterator_last(grid, X_DIM)))
        {
            x++;
        }
        else
        {
            y++;
            x = grid_iterator_first(grid, X_DIM);
        }
    }
    else if (y > grid_iterator_first(grid, Y_DIM) && y < (grid_iterator_last(grid,
```

```

    {
        if (x == grid_iterator_first(grid, X_DIM))
        {
            x = grid_iterator_last(grid, X_DIM);
        }
        else
        {
            y++;
            x = grid_iterator_first(grid, X_DIM);
        }
    }
else if (y == grid_iterator_last(grid, Y_DIM))
{
    if (x < grid_iterator_last(grid, X_DIM))
    {
        x++;
    }
    else /* this is the end */
    {
        x = grid_iterator_last(grid, X_DIM) + 1;
        y = grid_iterator_last(grid, Y_DIM) + 1;
    }
}

grid->current_tick[X_DIM] = x;
grid->current_tick[Y_DIM] = y;
UPDATE_CURRENT_VALUES(grid);
return OK;
}

int grid_guard_after(const grid *grid)
{
    /* true if after last guard */
    int dim, result;
    REQUIRE("grid_not_null", grid != NULL);
    result = (grid->current_tick[X_DIM] > (grid_iterator_last(grid, X_DIM)));

    for (dim = Y_DIM; dim <= grid->space_dimensions; dim++)
    {
        result = result && (grid->current_tick[dim] > (grid_iterator_last(grid, di
    }

```

```

    return result;
}

/* node retrieval */

int grid_item(const grid *grid, grid_node **node)
{
    /* return grid node at current iterator position */
    int dim;
    REQUIRE("grid_not_null", grid != NULL);

    grid_node_create(node);

    for (dim = T_DIM; dim <= grid->space_dimensions; dim++)
    {
        (*node)->tick[dim] = grid->current_tick[dim];
        (*node)->value[dim] = grid->current_value[dim];
    }

    (*node)->source_grid = grid;
    if (grid_node_is_internal((*node)))
    {
        (*node)->order = node_lexicographic_order((*node));
    }

    ENSURE("node_is_internal", grid_node_is_internal((*node)));
    return OK;
}

int grid_plain_item(const grid *grid, grid_node **node)
{
    /* return grid node in plain iteration,
       basically the same but with no postcondition */
    int dim;
    REQUIRE("grid_not_null", grid != NULL);

    grid_node_create(node);

    for (dim = T_DIM; dim <= grid->space_dimensions; dim++)

```



```

    {
        (*node)->tick[dim] = grid->current_tick[dim];
        (*node)->value[dim] = grid->current_value[dim];
    }

    (*node)->source_grid = grid;
    if (grid_node_is_internal((*node)))
    {
        (*node)->order = node_lexicographic_order((*node));
    }
    return OK;
}

int grid_loose_item(const grid *grid, int x, int y, grid_node **node)
{
    /* return a loose item, should be dimension independent */
    REQUIRE("grid_not_null", grid != NULL);

    grid_node_create(node);

    (*node)->tick[T_DIM] = grid->current_tick[T_DIM];
    (*node)->value[T_DIM] = grid->current_value[T_DIM];

    (*node)->tick[X_DIM] = x;
    (*node)->value[X_DIM] = TICK_TO_VALUE(grid, x, X_DIM);

    (*node)->tick[Y_DIM] = y;
    (*node)->value[Y_DIM] = TICK_TO_VALUE(grid, y, Y_DIM);

    (*node)->source_grid = grid;

    if (grid_node_is_internal((*node)))
    {
        (*node)->order = node_lexicographic_order((*node));
    }

    return OK;
}

int grid_focus_item(const grid *grid, grid_node **node)
{

```

```

int dim;
/* return grid node corresponding to focus position */
REQUIRE("grid_not_null", grid != NULL);

grid_node_create(node);

for (dim = T_DIM; dim <= grid->space_dimensions; dim++)
{
    (*node)->tick[dim] = grid->focus_tick[dim];
    (*node)->value[dim] = grid->min_value[dim] +
                        ((double)(*node)->tick[dim]) * grid->delta[dim];
}

(*node)->source_grid = grid;

if (grid_node_is_internal((*node)))
{
    (*node)->order = node_lexicographic_order((*node));
}

ENSURE("order_is_ge_zero", (*node)->order >= 0);
ENSURE("node_is_internal", grid_node_is_internal((*node)));
ENSURE("correct_x_value", APPROX_EQUAL((*node)->value[X_DIM], grid->focus[X_DIM]));
ENSURE("correct_y_value", APPROX_EQUAL((*node)->value[Y_DIM], grid->focus[Y_DIM]));
return OK;
}

int grid_node_neighbour(const grid *grid, int s_pos, const grid_node *src, grid_node *neigh)
{
    int x = 0, y = 0;
    int dim;
    /* return neighbour of src according to s_pos */
    REQUIRE("grid_not_null", grid != NULL);
    REQUIRE("src_node_not_null", src != NULL);
    REQUIRE("valid_space_position", s_pos >= XY && s_pos <= XPYP);

    STANDARD_CREATE(neigh, grid_node);

```

```

position_shift(s_pos, src->tick[X_DIM], src->tick[Y_DIM], &x, &y);

(*neigh)->tick[X_DIM] = x;
(*neigh)->tick[Y_DIM] = y;
(*neigh)->tick[T_DIM] = src->tick[T_DIM];

for (dim = T_DIM; dim <= grid->space_dimensions; dim++)
{
    (*neigh)->value[dim] = grid->min_value[dim] + ((double)(*neigh)->tick[dim]

(*neigh)->source_grid = grid;

if (grid_node_is_internal((*neigh)))
{
    (*neigh)->order = node_lexicographic_order((*neigh));
}

ENSURE("neighbour_created", (*neigh) != NULL);
ENSURE("grid_set", ((*neigh)->source_grid == grid));
ENSURE("valid_order", (*neigh)->order >= 0);
return OK;
}
/* end -- grid.c */

#endif //PremiaCurrentVersion

```