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 "
href../../../common/math/highdim_solver/cps_grid_node_h_src.pdfcps_grid_node.h"
#include "
href../../../common/math/highdim_solver/cps_types_h_src.pdfcps_types.h"
#include "
href../../../common/math/highdim_solver/cps_utils_h_src.pdfcps_utils.h"
#include "
href../../../common/math/highdim_solver/cps_assertions_h_src.pdfcps_assertions.h

int grid_node_create(grid_node **node)
{

  STANDARD_CREATE(node, grid_node);
  return OK;
}

int grid_node_destroy(grid_node **node)
{

  STANDARD_DESTROY(node);
  return OK;
}

int grid_node_is_boundary(const grid_node *node)
{
  /* check if node is boundary in space */
  const grid *grid;
```

```
  int dim, result;
  REQUIRE("node_not_null", node != NULL);

  grid = node->source_grid;

  result =
    grid_node_is_left_boundary(node, X_DIM) || grid_node_is_right_boundary(node,

  for (dim = Y_DIM; dim <= grid->space_dimensions; dim++)
    {
      result = result || (grid_node_is_left_boundary(node, dim)
                          || grid_node_is_right_boundary(node, dim));
    }
  return result;
}

int grid_node_is_external(const grid_node *node)
{
  int dim;
  int result = 0;
  const grid *grid
  /* check if node is external to current grid */
  REQUIRE("node_not_null", node != NULL);


  grid = node->source_grid;

  for (dim = X_DIM; dim <= grid->space_dimensions; dim++)
    {
      result = result ||
               ((node->tick[dim] < 0 ||
                 (node->tick[dim] >= grid->ticks[dim])));
    }

  return result;
}

int grid_node_is_internal(const grid_node *node)
{

  int dim;
```

```
  const grid *grid ;
  int result = 1;

  /* check if node is internal to current grid */
  REQUIRE("node_not_null", node != NULL);

  grid = node->source_grid;
  for (dim = X_DIM; dim <= grid->space_dimensions; dim++)
    {
      result = result &&
               ((node->tick[dim] >= grid_iterator_first(grid, dim)) &&
                (node->tick[dim] <= grid_iterator_last(grid, dim)));
    }

  return result;
}

int grid_node_is_left_boundary(const grid_node *node, int dim)
{
  int result;
  /* check if node is left boundary for given dimension */
  REQUIRE("node_not_null", node != NULL);
  REQUIRE("node_has_grid", node->source_grid != NULL);
  REQUIRE("valid_dimension", dim >= X_DIM && dim <= node->source_grid->space_dim

  result = ((node->tick[dim] == 0)
            && (grid_iterator_first(node->source_grid, dim) == 1));

  return result;
}

int grid_node_is_right_boundary(const grid_node *node, int dim)
{
  /* check if node is right boundary for given dimension */
  int result;
  REQUIRE("node_not_null", node != NULL);
  REQUIRE("valid_dimension", dim >= X_DIM && dim <= node->source_grid->space_dim

  result = ((node->tick[dim] == node->source_grid->ticks[dim] - 1)
            && (node->tick[dim] == grid_iterator_last(node->source_grid, dim) +
```

```
  return result;
}

int grid_node_is_guard(const grid_node *node)
{
  /* check if node is a guard */
  int dim;
  int result = 0;
  const grid *grid
  REQUIRE("node_not_null", node != NULL);
  REQUIRE("grid_is_set", node->source_grid != NULL);

  grid = node->source_grid;


  for (dim = X_DIM; dim <= grid->space_dimensions; dim++)
    {
      if (grid->current_iterator[dim] == ITER_CORE)
        {
          result = result || (node->tick[dim] == grid_iterator_first(grid, dim)
                              node->tick[dim] == grid_iterator_last(grid, dim));
        }
    }

  return result;
}

int grid_node_is_initial(const grid_node *node)
{
  /* check if node is initial */
  REQUIRE("node_not_null", node != NULL);

  return (node->tick[T_DIM] == 0);
}

int grid_node_is_final(const grid_node *node)
{
  /* check if node is on final time line */
  REQUIRE("node_not_null", node != NULL);

  return (node->tick[T_DIM] == node->source_grid->ticks[T_DIM]);
```

```
}

int grid_node_time_forth(grid_node *node)
{
  const grid *grid ;
  /* move time a step forth */
  REQUIRE("node_not_null", node != NULL);

  grid = node->source_grid;

  node->tick[T_DIM]++;
  node->value[T_DIM] = grid->min_value[T_DIM] + (double)node->tick[T_DIM] * grid
  return OK;
}

int grid_node_time_back(grid_node *node)
{
  const grid *grid;
  /* move time a step back */
  REQUIRE("node_not_null", node != NULL);
  REQUIRE("not_first", node->tick[T_DIM] > 0);

  grid = node->source_grid;

  node->tick[T_DIM]--;
  node->value[T_DIM] = grid->min_value[T_DIM] + (double)node->tick[T_DIM] * grid
  return OK;
}

/* end -- grid_node.c */

#endif //PremiaCurrentVersion
```