NaNChecker

Thomas Radke

Date: 2007/04/27 12:34:54

Abstract

Thorn NaNChecker reports NaN values found in CCTK grid variables.

1 Purpose

The NaNChecker thorn can be used to analyze Cactus grid variables (that is grid functions, arrays or scalars) of real or complex data type for NaN (Not-a-Number) and (on availability of `finite(3)`) infinite values. Grid variables can be periodically checked, or a call can be inserted into a thorn to check at a specific point.

This thorn is a utility thorn, designed to be used for debugging and testing code for uninitialised variables, or for variables which become corrupted during a simulation, for example following a division by zero or illegal memory usage.

On many architectures, uninitialised variables will be given the value zero, and simulations using such variables will seemingly run perfectly well. However, not only is it dubious programming practise to assume such behaviour, but also moving to a new machine may well cause pathalogical problems (for example, with Alpha processors used in Compaq or Cray machines). It is thus recommended to test codes periodically with the NaNChecker, and to fix any problems as soon as they are seen.

2 Periodic Testing

Periodic testing of variables can easily be achieved by adding NaNChecker to the `ActiveThorns` parameter, and setting the parameters

`NaNChecker::check_every`, `NaNChecker::check_after`, and `NaNChecker::check_vars`

to the required values. (For most testing purposes these can be set to `1`, `0`, and "all" respectively).

The NaNChecker then registers a routine at **CCTK_ANALYSIS** which checks at every `NanChecker::check_every` iteration – starting at iteration number `NaNChecker::check_after` – all the variables listed in `NaNChecker::check_vars` for NaN or infinite values (depending on `NaNChecker::check_for`) and — if such a value is found — performs an action as specified in `NaNChecker::action_if_found`.

Currently these actions can be to

- `just_warn` (the default)

  just print a level 1 warning message telling you where NaNs/Infs were found and how many (for grid array variables).
  If the keyword parameter `verbose` is set to `"all"` then for each grid array it will also print the grid indices (in Fortran notation) and the physical coordinates for all NaN/Inf elements found. You can limit the number of such warnings by setting the `NanChecker::report_max` parameter.

- `terminate`

  also set the CCTK termination flag so that Cactus will stop the evolution loop and gracefully terminate at the next time possible (giving you the choice of outputting the data from the last evolution timestep),

- `abort`

  print the warning message(s) and immediately terminate Cactus after checking all variables from `NaNChecker::check_vars` by a call to `CCTK_Abort()`

By default, the current timelevel of the variables given in `NaNChecker::check_vars` will be checked. This can be overwritten by an optional string `[timelevel=<timelevel>]` appended to the variable/group name. For example, to apply the NaNChecker to timelevel 0 of the variable grid::x, timelevel 1 of `grid::y` and timelevel 2 of `grid::z` you would use the parameter

```
NaNChecker::check_vars = "grid::x grid::y[timelevel=1] grid::z[timelevel=2]"
```

3 Tracking and Visualizing NaNs Positions

The NaNChecker thorn can also mark the positions (in grid index points) of all the NaNs found for a given list of CCTK grid functions in a mask array and save this array to an HDF5 file.

The mask array is declared as a grid function `NaNChecker::NaNmask` with data type INTEGER. Each bit i in an integer element is used to flag a NaN value found in grid function i at the corresponding grid position (the counting for i starts at 0 and is incremented for each grid function as it appears in `NaNChecker::check_vars`). Thus the NaN locations of up to 32 individual grid functions can be coded in the NaNmask array.

In order to activate the NaNmask output you need to set the parameter `NaNChecker::out_NaNmask` to `"yes"` (which is already the default) and have the IOHDF5 thorn activated.

The NaN locations can be visualized with OpenDX. An example DX network `VisualizeNaNs.net` and a sample NaNmask HDF5 output file `NaNmask.h5` are available via anonymous CVS from the NumRel CVS server:

```
# this is for (t)csh; use export CVSROOT for bash
setenv CVSROOT :pserver:cvs_anon@cvs.aei.mpg.de:/numrelcvs

# CVS pserver password is 'anon'
cvs login
cvs checkout AEIPhysics/Visualization/OpenDX/Networks/Miscellaneous
```

4 NaNChecker API

Thorn NaNChecker also provides a function API which can be used by other code to invoke the NaNChecker routines to test for NaN/Inf values or to set NaN values for a list of variables:

C API

```
int NaNChecker_CheckVarsForNaN (const cGH *cctkGH,
                                int report_max,
                                const char *vars,
                                const char *check_for,
                                const char *action_if_found);

int NaNChecker_SetVarsToNaN (const cGH *cctkGH,
                             const char *vars);
```

Fortran API

```
call NaNChecker_CheckVarsForNaN (ierror, cctkGH, report_max,
                                 vars, check_for, action_if_found)

                                 integer ierror
```

```
                                    CCTK_POINTER cctkGH
                                    integer report_max
                                    character*(*) vars
                                    character*(*) check_for
                                    character*(*) action_if_found

call NaNChecker_SetVarsToNaN (ierror, cctkGH, vars)

                                  integer ierror
                                  CCTK_POINTER cctkGH
                                  character*(*) vars
```

The `report_max, check_vars, check_for` and `action_if_found` arguments have the same semantics as their parameter counterparts.
If `action_if_found` is given as a NULL pointer (C API) or as an empty string (Fortran API) the routine will be quiet and just return the number of NaN values found.

The C function `NaNChecker_CheckVarsForNaN()` returns the total number of NaN/Inf values found, `NaNChecker_SetToNaN()` returns the total number of variables set to NaN; this return value is stored in the `ierror` argument for the corresponding fortran wrapper routines.